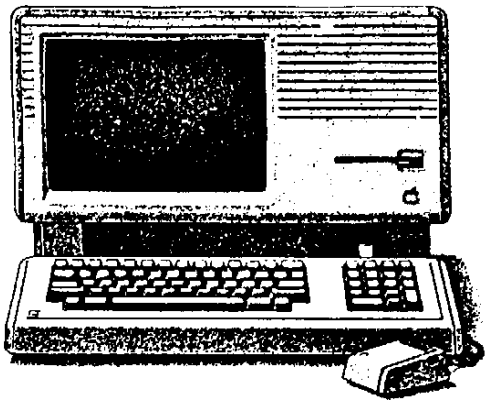
Document # 338

Apple Lisa Information



FILE NAME

Pascal Development System Manual

DISK #

COMMENTS

Bill Schottstaedt (Feb 82) (See Letter of 24 Oct 94)

David T. Craig
736 Edgewater, Wichita, Kansas 67230
(316) 733-0914

"LisaPascalDevManFeb82_000.PICT" 97 KB 2000-12-12 dpi: 72h x 72v pix: 1421h x 1927v

THE PASCAL DEVELOPMENT SYSTEM MANUAL

Bill Schottstaedt
Ext. 2379
16-Feb-82

Pages: 108

"LisaPascalDevManFeb82_000A.PICT" 8 KB 2000-12-12 dpi: 72h x 72v pix: 668h x 1113v

The Pascal Development System Manual

16 Feb 82

People names contained within:

- Bill Schottstaedt (cover author) - Rich Castro (LisaTest, p. 91)
- Ken Friedenbach (p. 12) 38 - Pete Cressman (p. 94)
- Rich (Page?) (p. 38) - Colin McMaster (p. 96)

Notes:

- new Lisa h/w p. 98
- LisaTest; p. 91
- ObjDiff; not in final dev sys (p. 90) BytеDiff
- Perform p. 93 - Coverage Analysis, p. 94
- PascalRef; windowed (p. 82)
- TLA Assembler on Apple II used to make Lisa Asm. (p. 23)
- Twiggy drive unreliable (p. 44)
- Symbolic debugger (not in final dev staff) p. 47
- Editor; cut/copy/paste keys (different) p. 60 SARAfont p. 61

Manual names contained within:

Jim Merritt p 13

- Pascal Language Reference Manual (p. 7) 12
- Development System Internal Documentation (p. 7) 46
- Lisa Production Tests (p. 91)

David T. Craig - Nov. 1994

"LisaPascalDevManFeb82_000B.PICT" 100 KB 2000-12-12 dpi: 72h x 72v pix: 1479h x 1847v

TABLE OF CONTENTS

The Monitor 1

The Pascal Compiler 7

The Linker 15

The Assembler 23

LisaBug 37

The Symbolic Debugger 47

The Filer 53

The Editors

The Lisa Editor 59
The UCSD Editor 63

Utilities 71

Segmentation and Intrinsic Unit Management
IUManager 72
ChangeSeg 74
SegMap 74

System Configuration
Configure 75
Contrast 75
SetSP 76
ChangeMem 76
Flip4 76
MoveSoroc 76

File Transfer
FileDiv 77
FileJoin 77

Source File Debugging
Diff 78
FindID 80
Pretty List 81
PascalRef 82

Object File Debugging
DumpOBJ 87
DumpHex 88
Patch 89
OBJDiff 90
ByteDiff 90
GXRef 90

"LisaPascalDevManFeb82_000C.PICT" 46 KB 2000-12-12 dpi: 72h x 72v pix: 1303h x 1895v

Hardware Debugging
LisaTest 91

Performance Measurement
Perform 93
Coverage Analysis 94

Document Compiler
Script 96

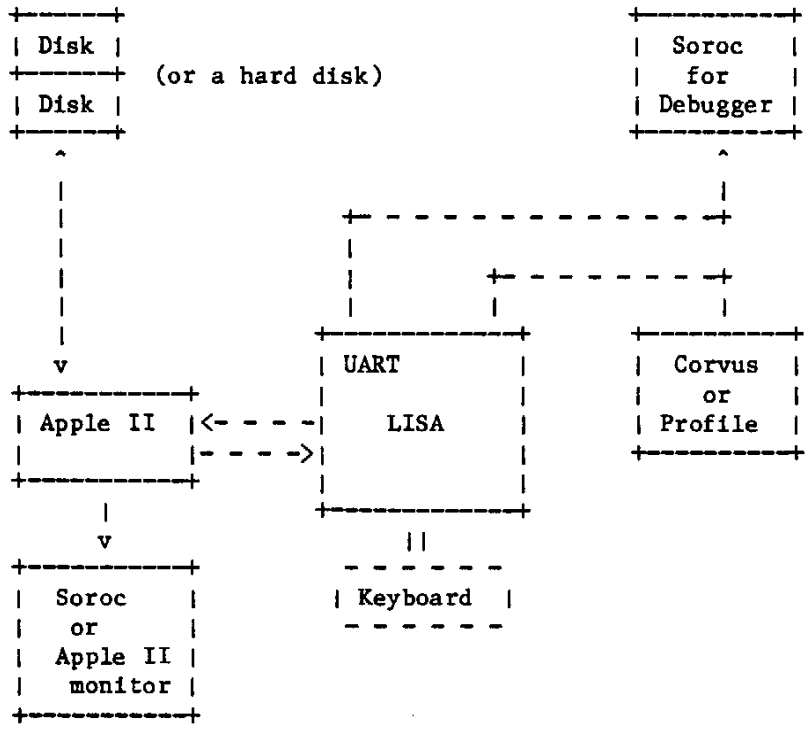
Other
Terminal Emulator 98

Error Messages 101

"LisaPascalDevManFeb82_000D.PICT" 13 KB 2000-12-12 dpi: 72h x 72v pix: 1318h x 669v

MONITOR

The Monitor is an operating system for the Lisa computer. Its user interface is patterned after that of the UCSD system on the Apple II. There are several possible system configurations. A standard one is:



The hard disk can be connected directly to the Lisa, or it can be accessed through the Apple II. It can also be omitted.

BOOTING THE MONITOR

To boot from a diskette based Apple II, first power up the Apple II with the male boot diskette in drive #4:. Insert the female boot in drive #5: and power up the Lisa. The female boot volume can also reside on a hard disk. SYSTEM.STARTUP on the male boot volume automatically executes MONBOOT, the program that starts up the Monitor on the Lisa. If you type space during the boot process, MONBOOT is not executed. If you type 'D' during the boot process, the debugging version of the Monitor is booted.

"LisaPascalDevManFeb82_001.PICT" 56 KB 2000-12-12 dpi: 72h x 72v pix: 1297h x 1994v

The Monitor comes up on the Lisa screen. If you want it to appear on the Apple II monitor or the Soroc connected to the UART port, change the MON.STARTUP program as follows:

1) for the Apple window: remove MON.STARTUP

2) for the Lisa window (the default): transfer MONSTART1.OBJ to MON.STARTUP

3) for the UART window: transfer MONSTART2.OBJ to MON.STARTUP

To move the Monitor around after booting, execute MOVESOROC. MOVESOROC simply asks you for the new source and destination for Monitor I/O: A(pple, L(isa, or U(art. Input always comes from the terminal to which output has been directed. WRITELNs used for debugging purposes appear on the Monitor screen, so you may not always want the application and Monitor screens to be together on the Lisa.

CONFIG.DATA tells the monitor how much RAM your system has. The default configuration assumes that you have a megabyte of RAM. For a 256K byte system, CONFIG.DATA should be a copy of NPC4.DATA. For a 512K byte system, use NPC8.DATA. NPC16.DATA is a copy of the default CONFIG.DATA, in case you ever need to back up. See CONFIGURE in the Utilities chapter if you want to change CONFIG.DATA to suit your own needs.

The monitor's keyboard driver supports the Lisa User Interface Keyboard layout. SHIFT [is {, SHIFT] is }, SHIFT . is >, and SHIFT , is <. NMI is the third key from the left in the upper row of the numeric keypad. Currently, the "4" key is backspace. The CODE key is in the upper left corner of the keyboard. CODE ; is ¦, CODE + is ~, CODE _ is \, and CODE " is `. ESCAPE is the upper left key of the numeric keypad. Control-S is the key in the upper right corner of the numeric keypad.

THE COMMAND LINE

The Monitor command line is:

Monitor: E(dit, C(ompile, F(ile, L(ink, A(ssemble, D(ebug, ? [0.1]

There are several hidden commands. Type ? to see them displayed.

E(dit	Lisa-style Editor
C(ompile	Pascal Compiler I-code generator
F(ile	Filer
I(intrinsic	Intrinsic Unit Linker (release 8.0 and beyond)
L(ink	Linker
A(ssemble	Assembler
D(ebug	Symbolic Debugger
M(acsBug	LisaBug (low level debugger)
G(enerate	Pascal compiler OBJ file generator
U(CSD	UCSD Editor
X(ecute	Execute a program or an EXEC file

"LisaPascalDevManFeb82_002.PICT" 92 KB 2000-12-12 dpi: 72h x 72v pix: 1314h x 2005v

The Monitor recognizes male volumes and logs them off-line so that they cannot be accidentally overwritten. The volume MEMORY: is always available. MEMORY: allows you to use the Lisa RAM as a file storage area. Of course, anything in MEMORY: is lost when the power is turned off, or the system is rebooted. MEMORY: is mounted as unit #4:, and its default size is 10 blocks. Its size can be changed by the Z(ero command in the Filer, or by the CHANGEMEM program described in the Utilities section of this manual.

When X(ecuting a program, the monitor searches for the program filename as follows:

```
Filename
Filename.OBJ
*Filename
*Filename.OBJ
Filename.TEXT   (* as an exec file *)
```

When you invoke a program from the Monitor command line (F for Filer.Obj, for example), the Monitor looks first at the MEMORY: volume.

EXEC FILES

EXEC files can be used on the Monitor. They must be created in the Editor (there is no M(ake command). To execute such a file,

```
X(ecute <filename>
```

If an object file exists with the same name as that of the EXEC file, the object file is executed. The first character of an EXEC file (a textfile) defines the termination character. The first occurrence of two terminators marks the end of the EXEC file. Certain portions of the system (the compiler, for example) terminate an EXEC file if an error is encountered. If you X(ecute a .TEXT file, the monitor assumes that the file is an EXEC file. EXEC files cannot be nested, nor can parameters be passed to them.

"LisaPascalDevManFeb82_003.PICT" 64 KB 2000-12-12 dpi: 72h x 72v pix: 1305h x 1936v

LOW MEMORY LAYOUT

From	To	Description
0000	00FF	Exception vectors (see 68000 manual)
0100	01FF	Memory configuration map (see page ##)
0200	0300	Free space for user assembly globals
0300	0341	KCS numerics status information
0342	03FF	Free space
0400	07FF	LisaBug Globals
0800	08FF	Boot stack
0900	0AFF	LisaGraf Globals
0B00	0Bxx	Unit Table
0C00	0CFF	Pointer array
0D00	0Dxx	Syscom, miscinfo
0E00	0EFF	String buffer
0F00	0FFF	Unused (reserved) space
1000	17FF	User code buffer
1800	3FFF	User Jump Table
4000		Heap Bottom

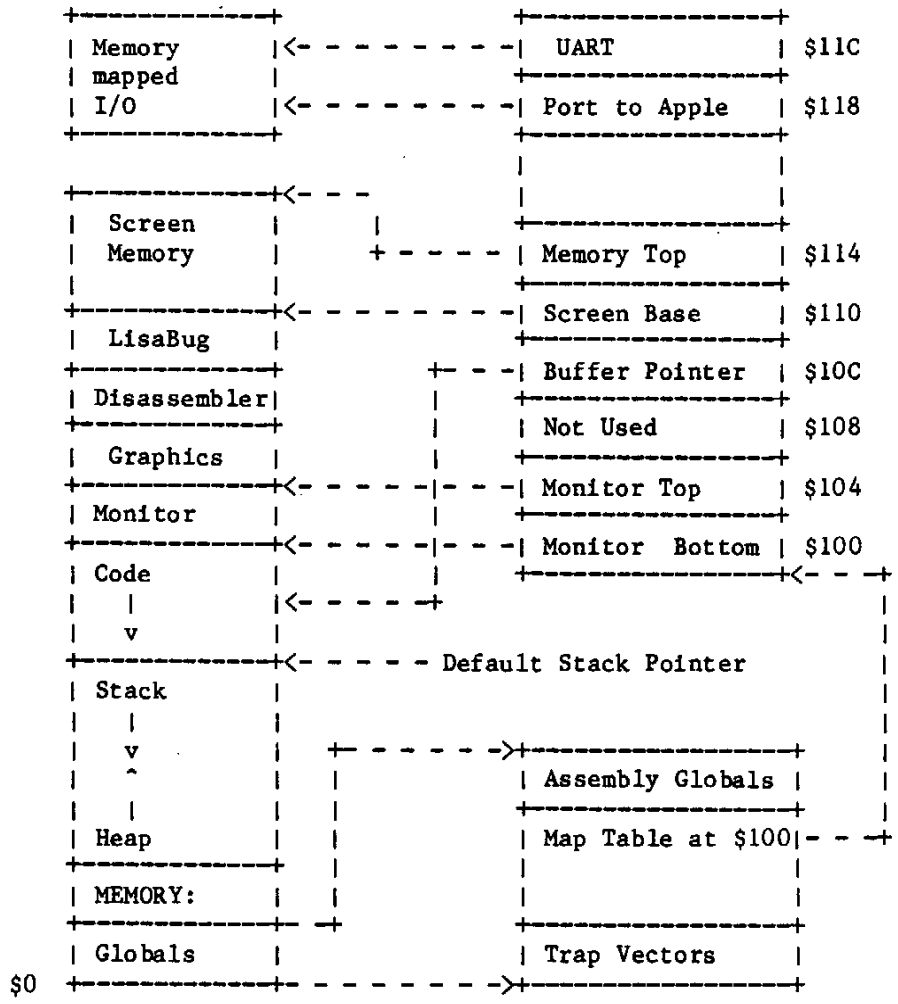
Registers	Description
A7	Stack Pointer
A6	Stack frame Pointer
A5	Global Data Pointer
A3-A4	Used for code optimization
A2-A0	Scratch
D0-D3	Scratch
D4-D7	Used for code optimization

Registers D3 and A2 may someday be used by the compiler for code optimization.

"LisaPascalDevManFeb82_004.PICT" 42 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1994v

MEMORY MAP

A set of very detailed memory maps can be found in the Linker chapter. We give below a general view of memory and a detailed view of the Monitor's Map Table.



"LisaPascalDevManFeb82_005.PICT" 48 KB 2000-12-12 dpi: 72h x 72v pix: 1301h x 1979v

THE MAP TABLE

Monitor bottom	$100
Monitor top	$104
	$108
Buffer pointer	$10C
Screen base	$110
Memory top	$114
Port to Apple II	$118
UART	$11C
Ptr to Lisabug jump table	$120
Ptr to GOTOXY	$124
Ptr to Soroc driver	$128
Ptr to soft break table	$12C
Ptr to UART driver	$130
Ptr to Corvus card	$134
Ptr to base of heap	$138
Default SP	$13C
Ptr to user's last A6	$140
Ptr to MEMORY:	$144
Ptr to Twiggy driver	$148
Ptr to hard disk Jump Table	$14C
Ptr to debug card	$150
Ptr to loader for IUs	$154
	$158
	$15C
	$160
	$164
Apple net	$168
Apple net	$16C

Many of these vectors can be changed by the CONFIGURE program described in the Utility section of this manual. The main vector of interest is the default stack pointer ($13C). The utility program SETSP can be used to change the default stack pointer value temporarily. CONFIGURE can be used to change it permanently. Unused addresses are reserved for future use by the Monitor.

"LisaPascalDevManFeb82_006.PICT" 46 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1990v

THE COMPILER

Files needed: COMPILER.OBJ
CODE.OBJ
MPASLIB.OBJ, NOFPLIB.OBJ, or IUPASLIB.OBJ

GENERAL INFORMATION

The compiler is split into two programs, COMPILER.OBJ and CODE.OBJ. COMPILER.OBJ (invoked by the Monitor's C(ompile option) parses the Pascal program text into semantically equivalent tree structures. CODE.OBJ (invoked by the G(enerate command) then turns these trees into 68000 code. The compiler follows the proposed ISO standard Pascal with some exceptions and extensions. A complete definition of Lisa Pascal can be found in the Pascal Language Reference Manual. The definition of I-code formats and MPaslib information can be found in the Development System Internal Documentation.

The compiler first asks for the

```
Input file -
```

The .TEXT extension is added, if necessary. In the following prompts, the bracketed text is used if you leave out that portion of the file names.

```
Listing file (<cr> for none) -
Output file [<input name>] [.I]
Debug file [<input name>] [.DBG] -
```

If you do not want a debug file created, type <ESC><cr>. <cr> always accepts the default setting. If you write both the .I file and the .DBG file to the same volume, use the [*] specification on the .I file to avoid space problems. The trouble arises when you have one large block on the volume. When the operating system allocates space for a file, it gives the file all of the largest block it can find unless you specify otherwise. If no other block of space exists and all of the existing block has been allocated to the .I file, you get a "no room on vol" error when the system attempts to open the .DBG file, even if there is plenty of room for both. The [*] specification tells the operating system to allocate only half of the largest available block to the file.

The Pascal run time support routines are in MPASLIB. If you do not need the floating point arithmetic routines, you can use NOFPLIB instead of MPASLIB. If you are using intrinsic units, use IUPASLIB.

"LisaPascalDevManFeb82_007.PICT" 83 KB 2000-12-12 dpi: 72h x 72v pix: 1308h x 1985v

COMPILER OPTIONS

$C+ or $C- - Turns code generation on (+) or off (-) on a procedure by procedure basis. The default is C+.

$D+ or $D- - If the $D option is on (the default), the compiler places procedure names in the object file. The object file is slightly larger, but LisaBug use becomes much more pleasant.

$DECL - Compile time variable declaration (conditional compilation). Compile time variables must be declared before they can be used (in $SETC), and all declarations must precede the first procedure or function definition in the program. The $DECL compiler option does not exist until the version 8.0 compiler.

$E filename - Starts logging compile time errors as they are encountered. This option is analogous to the $L option.

$ELSEC - Conditional compilation.

$ENDC - Conditional compilation.

$I filename - Includes the file 'filename' in the compilation. The filename cannot begin with a '+' or a '-'.

$IFC - Conditional compilation.

$L filename - Starts making a listing of the compilation in file 'filename'. If a listing is already in progress, that file is closed and saved before the new listing file is opened.

$L+++ or $L--- - The first +/- turns listing on (+) or off (-) during the first pass. The second +/-, if present, turns on or off the listing with object code offsets during the second pass. The third +/-, if present, controls the production of an interlisting during the second pass.

$R+ or $R- - Turns range checking on (+) or off (-). Currently, range checking is done in assignment statements, on array indexes, and for string value parameters. The default is $R+.

$S segment - Starts putting code modules into the segment named 'segment'. The default segment (' ') holds the main program and all built-in support code. All other code can be placed in any segment.

$SETC - Compile time variable declaration and assignment.

$U filename - Searches the file 'filename' for any subsequent units.

"LisaPascalDevManFeb82_008.PICT" 80 KB 2000-12-12 dpi: 72h x 72v pix: 1318h x 1990v

$X+ or $X- - Turns stack expansion code on (+) or off. The default is $X+.

$%+ or $%- - Allows the use of percent signs as legal characters in identifier names. The default is $%-. The % option should not be used by normal applications.

PACKING INFORMATION

Packed records are very expensive in terms of the number of bytes of code generated by the compiler to reference a field of a packed record. In general, you should avoid packing records unless there are many more instances of a particular record than there are references to it.

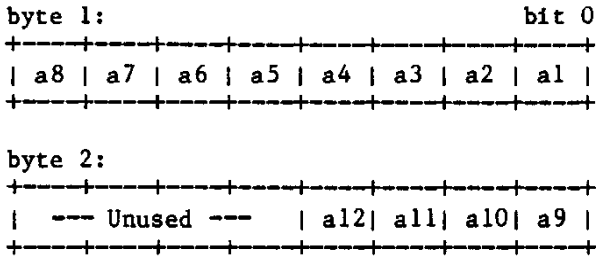
Packed arrays are also code-expensive, with one exception. Packed arrays of char are treated as a special case, and the code associated with them is compact.

To paraphrase von Neumann, anyone who needs to know the details of the packing algorithms is in a state of sin, but the following is provided for the sake of completeness.

Elements of packed arrays are stored with multiple values per byte whenever more than one value can be fit into a byte. This only happens when the values require 4 bits or less. Values requiring 3 bits are stored into 4 bits.

The first value in a packed array is stored in the lowest numbered bit position of the lowest addressed (most significant) byte. Subsequent values are stored in the next available higher numbered bit positions within that byte. When the first byte is full, the same positions are used in the next higher addressed byte. Consider the following examples:

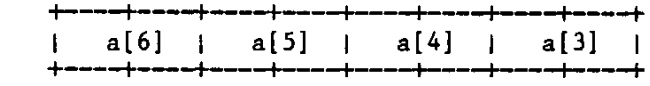
a: PACKED ARRAY[1..12] OF BOOLEAN



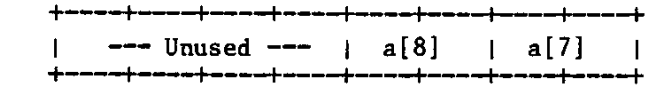
"LisaPascalDevManFeb82_009.PICT" 72 KB 2000-12-12 dpi: 72h x 72v pix: 1302h x 1994v

b: PACKED ARRAY[3..8] OF 0..3

byte 1:

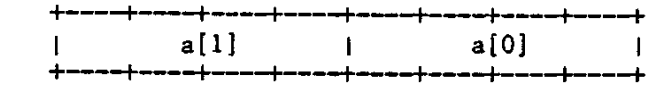


byte 2:

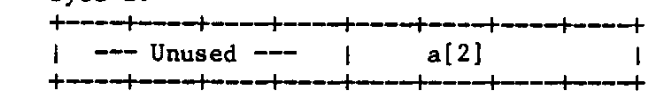


c: PACKED ARRAY[0..2] OF 0..7
or
PACKED ARRAY[0..2] OF 0..15

byte 1:



byte 2:



You can use the @ operator to poke around inside any packed value and thereby discover what the packing algorithm (probably) is. For example, to get the data given above, you can use a program like the following:

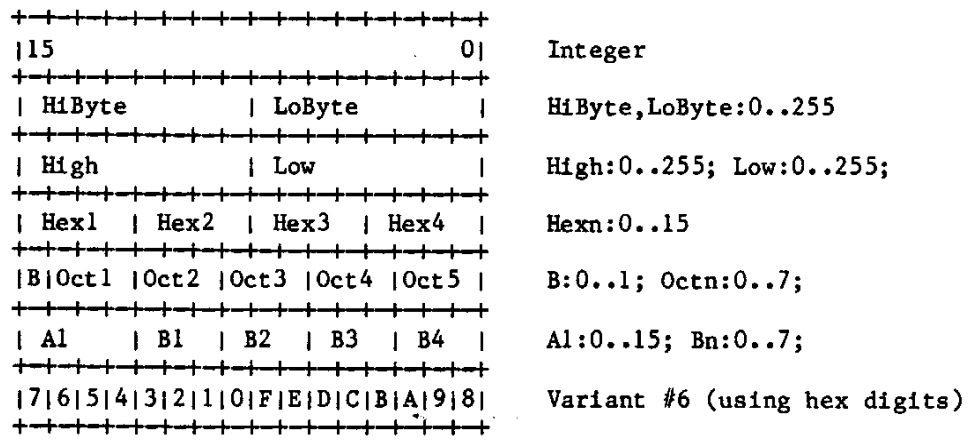
```
Program Test;
Var i:integer;
    p:^integer;
    boolArr:packed array [1..12] of boolean;
Begin
boolArr[1]:=true;       (* find out where 1st bit is put *)
for i:=2 to 12 do boolArr[i]:=false;
p:=@boolArr;
WriteLn('equiv word is ',p^);
                        (* write the packed array as an integer *)
End.
```


"LisaPascalDevManFeb82_010.PICT" 46 KB 2000-12-12 dpi: 72h x 72v pix: 1315h x 1985v

Consider also the following program fragment:

```
BITE = 0..255;
WORDSWAP = PACKED RECORD
                CASE INTEGER OF
                    0:(HWord:INTEGER);
                    1:(HiByte,LoByte:BITE);
                    2:(High:BITE;
                       Low:BITE);
                    3:(Hex1,Hex2,Hex3,Hex4:0..15);
                    4:(Bool:0..1;
                       Oct1,Oct2,Oct3,Oct4,Oct5:0..7);
                    5:(A1:0..15;
                       B1:0..7;
                       B2:0..7;
                       B3:0..7;
                       B4:0..7);
                    6:(Bin:PACKED ARRAY[0..15] OF 0..1)
                END;
```

Each variant gets packed into 16 bits. The question then is, where in the 16 bits do the various portions of the variants get placed:



"LisaPascalDevManFeb82_011.PICT" 52 KB 2000-12-12 dpi: 72h x 72v pix: 1314h x 1987v

LISA PASCAL AND APPLE PASCAL

Lisa and Apple Pascal are quite similar. We give below a list of the major differences, and a section of hints for translation from Apple to Lisa Pascal. Full details can be found in the Pascal Language Reference Manual.

EXTENSIONS TO APPLE PASCAL

@ Operator
CASE OTHERWISE Clause
POINTER function
Hexadecimal constants
DISPOSE
ORD4 function
Global GOTO
Parametric Procedures and Functions

DELETIONS FROM APPLE PASCAL

Initialization block in UNIT declaration
PWROFTEN, TREESEARCH, BYTESTREAM, WORDSTREAM, KEYBOARD
Extended comparisons
Some Compiler options
SEGMENT Procedures and Functions

REPLACEMENTS FOR APPLE PASCAL FEATURES

Long Integers -- 32 bit integers
Scan -- ScanEq and ScanNe
TURTLEGRAPHICS and APPLESTUFF -- LisaGraf

TRANSLATION FROM APPLE PASCAL TO LISA PASCAL

Translation of Apple Pascal programs is usually not very difficult. The following hints may be of use to you if you find yourself saddled with the translation task. Thanks to Ken Friedenbach for the hints!

MOVELEFT(Source_Buf[i],Dest_Buf[k],n) can be translated into:

```
FOR LocalI:=0 TO n-1 DO Dest_Buf[LocalI+k]:=Source_Buf[LocalI+i];
```

It may be necessary to declare the local integer used as the FOR loop control variable.

MOVERIGHT(Source_Buf[i],Dest_Buf[k],n) becomes:

```
FOR LocalI:=n-1 DOWNTO 0 DO Dest_Buf[k+LocalI]:=Source_Buf[i+LocalI];
```


"LisaPascalDevManFeb82_012.PICT" 62 KB 2000-12-12 dpi: 72h x 72v pix: 1313h x 1985v

FILLCHAR(Buf[i],n,Ch) becomes:

```
FOR LocalI:=0 TO n-1 DO Buf[i+LocalI]:=ch;
```

i:=SCAN(n,<>ch,Buf[k]) becomes:

```
LocalI:=0;
IF n>0 THEN
  WHILE (LocalI<n) AND (Buf[k+LocalI]=ch) DO LocalI:=LocalI+1
ELSE
  WHILE (LocalI>n) AND (Buf[k+LocalI]=ch) DO LocalI:=LocalI-1;
i:=LocalI;
```

If SCAN is looking for =ch, just substitute <>ch in the loops above.

READ(KEYBOARD,ch) becomes:

```
UNITREAD(2,ChArr,1);
ch:=ChArr[0];
```

where chArr=packed array [0..1] of char.

EOLN(KEYBOARD)

can check the character read above. If ch=CHR(13) then EOLN is true.

KEYPRESS

is NOT UNITBUSY(2).

Strings must be given a length, non-local EXITs must be replaced with GOTOs.

ClearScreen and other such functions can be handled by Jim Merritt's CUSTOMIO unit. ClearScreen on the Lisa is presently WRITE(CHR(27),CHR(42));

If underbars are used in the Apple Pascal program, they must be used consistently (they are ignored by the Apple Pascal Compiler!).

If the Apple Pascal units have code in the intialization block, put it in a procedure called at the beginning of the program.

"LisaPascalDevManFeb82_013.PICT" 51 KB 2000-12-12 dpi: 72h x 72v pix: 1301h x 1984v

To force segments to be resident, build a chain of dummy procedure calls that forces the loader to keep them all in core. The main program then becomes a procedure called by the top of the chain. Say we have 3 segments called SEG1, SEG2, and SEG3, and have put our main program into a procedure named MAIN_PROGRAM. We can now force everything to be memory resident by adding the following procedures:

```
(*$S SEG1*)
Procedure Kludge3;
BEGIN
Main_Program;
END;

(*$S SEG2*)
Procedure Kludge2;
BEGIN
Kludge3;
END;

(*$S SEG3*)
Procedure Kludge1;
BEGIN
Kludge2;
END;

(*$S            *)
BEGIN
Kludge1;
END.           (* end of main program *)
```


"LisaPascalDevManFeb82_014.PICT" 35 KB 2000-12-12 dpi: 72h x 72v pix: 1312h x 1995v

THE LINKER

Files needed: LINKER.OBJ or IULINKER.OBJ

GENERAL INFORMATION

The Linker combines object files. Its input consists of commands and object files. Its output consists of object files, link-map information, and error messages. Partial links are allowed. The output of the compiler must be linked with some version of PASLIB.OBJ before it can be executed. Other object files, including libraries, partial links, and object files produced by the Assembler, can also be linked into the output object file.

The Intrinsic Unit Linker (IULINKER.OBJ) expects to find the file *INTRINSIC.LIB even if you are not using any intrinsic units. LINKER.OBJ ('The Linker' in this chapter) expects to find LOADER.IMAGE somewhere on the system.

LINKER PROMPTS

The linker first prompts for the names of the input files:

```
Input file [.OBJ] -
```

It continues to ask for input files until you type <cr>. The next request is for the

```
Listing file -
```

Type <cr> if you don't want any listing file. The last request is for the name of the

```
Output file [.OBJ] -
```

LINKER COMMAND FILES

The Linker can read commands from a text file. At any time you can switch to such a file by typing '<' followed by the name of the file in which the commands reside. If there is a blank line in the file, the Linker assumes that this line is equivalent to the <cr> typed to end input file input. The line after the blank line (if any) is the listing file name, and the line after that is the output file name. These two files need not be given in the command file.

"LisaPascalDevManFeb82_015.PICT" 66 KB 2000-12-12 dpi: 72h x 72v pix: 1309h x 1986v

LINKER OPTIONS

Linker options can be entered at any time in response to the prompt for an input file. The options do not have any effect until the link begins. In particular, segment names cannot be mapped to several different names.

The Intrinsic Unit Linker has the following options:

+A Alphabetical listing of symbols. The default is -A.

+D Debug information. The default is -D.

+H num +H sets the maximum amount of heap space the Operating System can give a program before allowing it to die. Here, as in the other options, 'num' can be either decimal or hexadecimal.

-H num -H sets the minimum amount of heap space needed by a program.

+L Location ordered listing of symbols. The default is -L. The location is the segment name plus offset.

+M fromName toName
+M maps all occurrences of the segment 'fromName' to the segment 'toName'. This allows you to map several small segments into a single larger segment. You can thereby postpone the segmentation decision until link time by using many segment names in the source code.

+P Production link. The default is -P. +P produces a 'production' .OBJ file. A production object file does not contain information used by the debugger and the linker, and intrinsic unit files do not contain a jump table. The production object file can be executed, but cannot be handled by the linker or the debugger.

+S num
+S sets the starting dynamic stack size to 'num'. The default is currently 10000.

+T num
+T sets the maximum allowed location of the top of the stack to 'num'. The default is 128K.

? Prints the options available and their current values.

The Linker has the following options:

? Print out the options and their current values

Q Use Quick_Load blocks in place of Executable blocks. The Monitor has never supported this option.

"LisaPascalDevManFeb82_016.PICT" 75 KB 2000-12-12 dpi: 72h x 72v pix: 1302h x 1993v

P Do a Physical (+P) or Logical (-P) link. +P is the default. The logical link uses the MMU's to map logical addresses into physical memory. The physical link maps all of memory linearly. A logically linked program is more sensitive to uninitialized pointer problems than a physicaly linked program. If a physical link is performed, the linker and the executable program it produced must execute with the default stack pointer set to the same location. The default stack pointer value is $80000.

THE LINKER OUTPUT FILE

If no errors occur during the link, the output file contains the result of the link. If all external references are resolved and a starting location is specified, the output file is an executable object file. You must link in MPASLIB.OBJ or its equivalent to resolve all external references.

ERROR MESSAGES

The Linker reacts in three general ways to dubious usage. It gives a warning message if some action cannot be performed. This kind of message can be distinguished from the others by carefully noting that it begins with:

*** Warning

In order to recover from the error, simply reenter the command correctly, and all will proceed as though no error had occurred.

An error that makes it impossible for the Linker to complete the link successfully causes a message that begins:

*** Error

The link process can be continued, however, so that any further problems can be discovered.

A fatal error causes the link to be terminated immediately and sends a message beginning with:

*** Fatal Error

See the section on errors for a complete list of the Linker error messages.

EXTERNAL NAMES

An external name is a symbolic entry point into an object module. All such names are visible at all times--there is no notion of the nesting level of an external name. External names can be either global or local. A local name begins with a $ followed by 1 to 7 digits. No other characters are allowed. A global name is any name which is not a local name.

The scope of a global name is the entire program being linked. Unsatisfied references to global names are allowed. Only one definition of a given global name may occur in a given link.

The scope of the local name is limited to the file in which it resides. When a partial link is done, global names are passed through to the output file unmodified, but local names are renamed so that no conflicts occur between local names defined in more than one file. All references to a given local name must occur within the same input file.

MODULE INCLUSION

The first file presented to the Intrinsic Unit Linker must be either a main program file to be linked, or an unlinked intrinsic unit file. You cannot have both intrinsic unit and main program files in a single link. All modules from a non-library file are included in the output file. Only those modules which are needed in the link, however, are taken from a library file. The Linker considers a module to be needed if:

1) it defines an unresolved global name, or

2) it is referenced by a module in the same library file that is included in the output file.

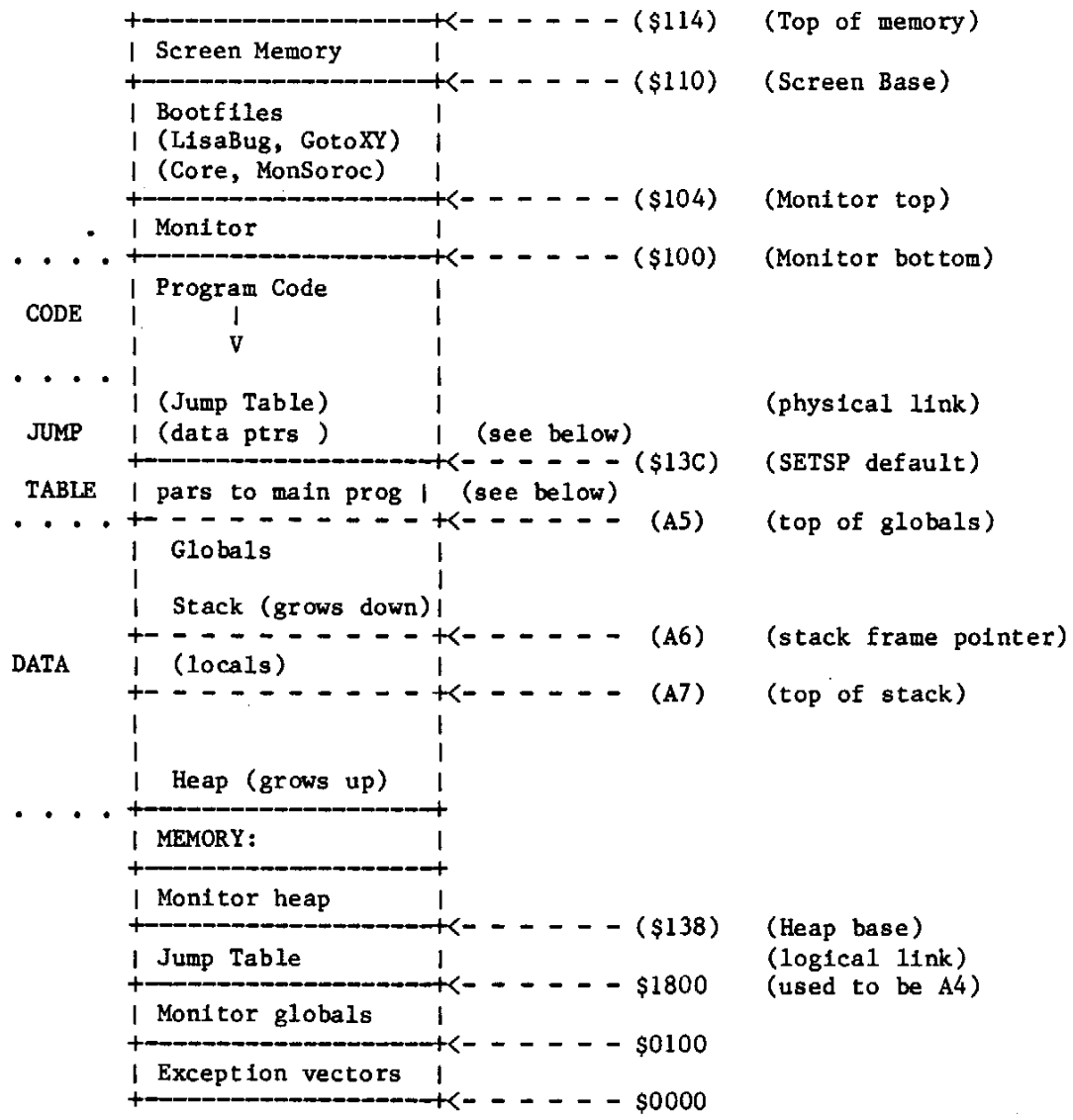
A module is not included simply because it references an already defined global name. Thus, the inclusion of a library module is dependent on the order in which files are specified to the Linker--the module must be specified after the modules that reference it. You can easily use an alternate module to one in a library by including the alternate prior to specifying the library file.

After linking an intrinsic object file and before referring to it in another link, ypou must update the segment and unit tables in *INTRINSIC.LIB with the IUMANAGER utility. IUMANAGER is described in the Utilities Chapter of this manual.

"LisaPascalDevManFeb82_018.PICT" 87 KB 2000-12-12 dpi: 72h x 72v pix: 1318h x 1996v

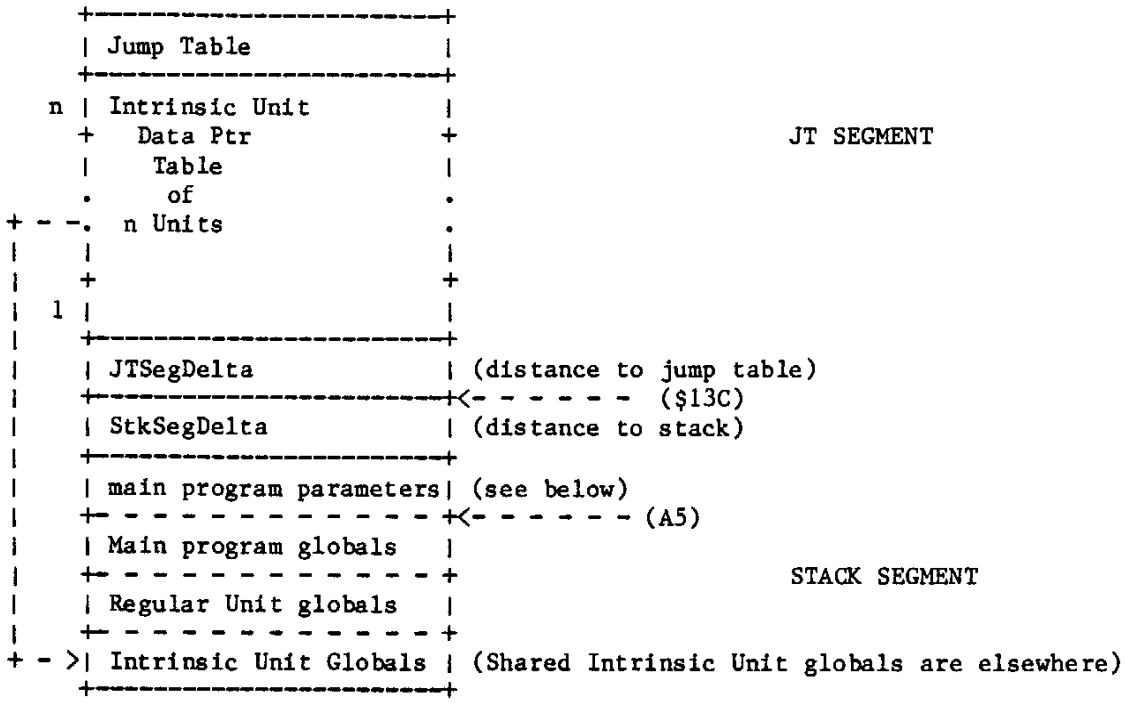
STRUCTURE OF AN EXECUTING PROGRAM

When a program is executing, the Lisa memory map is:



"LisaPascalDevManFeb82_019.PICT" 48 KB 2000-12-12 dpi: 72h x 72v pix: 1314h x 1963v

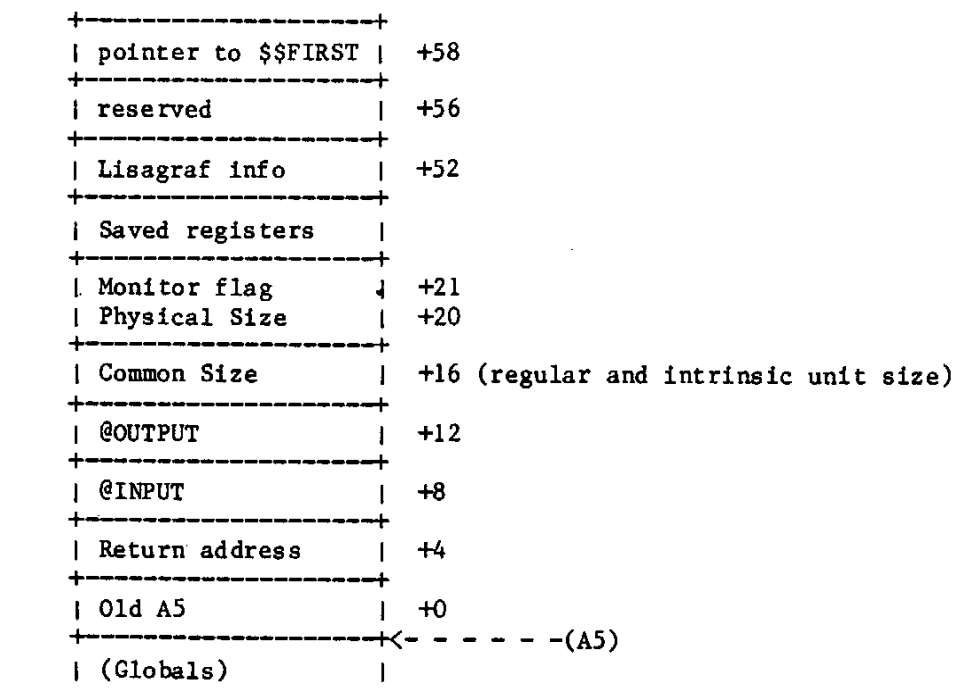
A physical link places the Jump Table above A5, but a logical or Intrinsic Unit link places the Jump Table at $1800 to free up A4 for code optimization. Even when placed at $1800, the Jump Table is logically above A5. The program globals are located below A5. The details of the portion of memory addressed by offsets from A5 is:



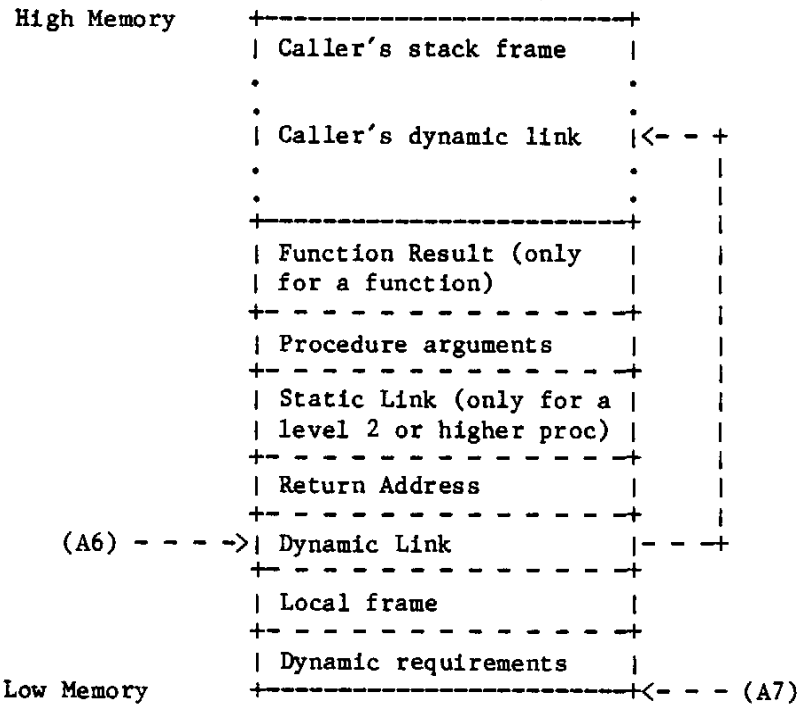
If the program is using shared intrinsic units, some of the intrinsic unit data pointers point to locations in the Shared Data Segment which contains global data used by all processes. The JT Segment is read-only and grows up. The stack segment is read-write and grows down.

"LisaPascalDevManFeb82_020.PICT" 53 KB 2000-12-12 dpi: 72h x 72v pix: 1337h x 1984v

The parameters to the main program are:



A6 is the stack frame pointer. The stack frame of a procedure is:



"LisaPascalDevManFeb82_021.PICT" 45 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1978v

THE 68000 ASSEMBLER

Files Needed: ASSEMBLER.OBJ
N68K.OPCODES
N68K.ERRORS

GENERAL INFORMATION

The Assembler is derived from the TLA Assembler on the Apple II. When invoked, it asks first for

```
Input file [.TEXT] -
```

You can reset the values of the options displayed, or give the name of the assembly source file. The next prompts are:

```
Listing file ( <CR> for none) -
Output file [inputname] [.OBJ[*] ] -
Symbols file [inputname] [.SYMBOLS] ( <ESC> for none) -
```

The symbols file is sometimes used during debugging, although the compiler D+ option provides a similar service with less hassle. The symbol table is discussed in more detail below in the section 'Communication with Pascal'.

If you specify a file, rather than PRINTER: or CONSOLE: as the listing file, it is probably wise to specify that the listing file take only half of the largest area on the volume by adding [*] to the file name. If this is not done, the first opened file may take up all the free area on the volume, and later attempts to open a file will fail. The assembler uses a temporary work file, so even if you do not ask for a listing file, the system may complain about not being able to find room on the volume. If you specify a size, be certain the size is not too small for the listing file.

If an error is encountered and the file N68K.ERRORS is on your prefix volume, the Assembler gives an error message as well as the error number. The error messages are also given in the Errors chapter of this manual.

The 68000 opcodes are described in the Motorola MC68000 Microprocessor User's Manual. The assembler has two variant mnemonics for branches (BHS for BCC and BLO for BCS). The variant names are more indicative of how the instruction is being used after unsigned comparisons. The default radix is decimal. It should be noted that the Assembler accepts generic instructions and assembles the correct form. The instruction ADD, for example, is assembled into ADD, ADDA, ADDQ, or ADDI, depending on the context.

```
ADD     D3,D5
```

becomes `ADDA    D3,D5`.

MOVE, CMP, and SUB are handled in a similar manner.

"LisaPascalDevManFeb82_023.PICT" 86 KB 2000-12-12 dpi: 72h x 72v pix: 1304h x 1973v

ASSEMBLER OPTIONS

The Assembler has three options:

M	toggles whether detailed Pass2 information is printed
S	toggles whether information about available space is printed
C	determines whether a .CODE or .OBJ file is created. +C produces a .CODE file (for male byte sex machines). -C (the default) produces a .OBJ file.

The current value of each option is displayed when the Assembler is invoked.

ASSEMBLER SYNTAX

$	= hex
@	= local label
_	is a legal identifier character
%	A legal identifier character except inside a macro definition. In a macro definition, %n is a reference to the nth parameter of the macro.
.	is a legal character
#	immediate operand
'	delimits strings
;	begins comments
*	current location

The size of an operation (byte, word, or long) is specified by appending either .B, .W, or .L to the instruction. The default operation size is word. To cause a short forward branch, append a .S to the instruction. The default branch size is Long.

Only the first eight characters of identifier names are meaningful to the assembler. The first character must be alphabetic; the rest must be alphanumeric, period, underbar, or percent sign.

Labels begin in column one. They can be followed by a colon, if you like. Local labels can be used to avoid using up the storage space required by regular labels. The local label stack can handle 21 labels at a time. It is cleared every time a regular label is encountered. Local labels in this assembler start with the character @.

"LisaPascalDevManFeb82_024.PICT" 64 KB 2000-12-12 dpi: 72h x 72v pix: 1314h x 1993v

All quantities are 32 bits in size unless constrained by the instruction. Expressions are evaluated from left to right with no operator precedence. Angle brackets can be used to control expression evaluation. The following operators are available:

+	unary or binary addition
-	unary minus or subtraction
~	ones complement (unary operator)
^	exclusive or
*	multiplication
/	division (DIV)
\	MOD
|	logical OR
&	logical AND
=	equal (used only by .IF)
<>	not equal (used only by .IF)

The following is a summary of the addressing mode syntax for the 68000:

Mode	Register	Syntax	Meaning	Extra Words
0	0..7	Di	Data direct	0
1	0..7	Ai	Address direct	0
2	0..7	(Ai)	Indirect	0
3	0..7	(Ai)+	Postincrement	0
4	0..7	-(Ai)	Predecrement	0
5	0..7	e(Ai)	Indexed	1
6	0..7	e(Ai,Ri)	Offset indexed	1
7	0	e	Absolute short address	1
7	1	e	Absolute long address	2
7	2	e	PC Relative	1
7	3	e(Ri)	PC Relative indexed	1
7	4	#e	Immediate	1 or 2

Notes:

1) The indexed and PC relative indexed modes are determined by the opcode.
2) The absolute address and PC relative address modes are determined by the type of the label (absolute or relative).
3) The absolute short and long address modes are determined by the size of the operand. Long mode is used only for long constants.
4) The number of extra words for immediate mode is determined by the opcode (.W or .L).

To specify which registers are affected by Move Multiple (MOVEM), specify ranges of registers with "-", and specify separate registers with "/". For example, to push registers D0 through D2, D4, and A0 through A4 onto the top of the stack:

```
MOVEM.L   D0-D2/D4/A0-A4,-(A7)
```


"LisaPascalDevManFeb82_025.PICT" 71 KB 2000-12-12 dpi: 72h x 72v pix: 1305h x 1971v

ASSEMBLER DIRECTIVES

The Assembler directives (pseudo-ops) are:

```
.PROC   <identifier>[,Expr]     begin procedure with Expr args
.FUNC   <identifier>[,Expr]     begin function with Expr args
.END                            end of entire assembly

.ASCII  '<character-string>'    place ASCII equivalents of chars in code
.BYTE   <value-list>            allocate a byte in code for each value
.BLOCK  <length>[,value]        allocate length bytes of value
.WORD   <value-list>            allocate a word for each value
.LONG   <value-list>            allocate a long word for each value
.ORG    <value>                 place next byte at <value>
.EQU    <value>                 set label equal to <value>

.MACRO  <identifier>            begin macro definition
.ENDM                           end macro definition

.IF     <expr>                  begin conditional assembly
.ELSE                           optional alternate to .IF block
.ENDC                           end conditional assembly

.DEF    <identifier-list>       make identifiers externally available
.REF    <identifier-list>       declare external identifiers that will be used

.LIST                           turn on assembly listing
.NOLIST                         turn off assembly listing
.PAGE                           issue a page feed in listing
.TITLE  '<title>'               title of each page in listing

.INCLUDE <filename>             insert <filename> into assembly
```

COMMUNICATION WITH PASCAL (.PROC and .FUNC)

Pascal programs can call assembly language procedures in a manner similar to that found in the UCSD system. The Pascal program declares the assembly language procedure or function to be EXTERNAL. If the assembly routine does not return a value, use .PROC. If .FUNC is used, space for the returned value is inserted on the stack just before the function parameters, if any. The amount of space inserted depends on the type of the function. A LongInt or Real function result takes two words, a Boolean result takes one word with the result in the high order byte, and other types take one word. In the following example, we link a bit-twiddling assembly language routine into a Pascal program. The Pascal host file is:

"LisaPascalDevManFeb82_026.PICT" 78 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1984v

```
PROGRAM BITTEST;
VAR I,J: INTEGER;
FUNCTION  Iand( i, j : INTEGER ) : INTEGER;
    EXTERNAL;             (* external = Assembly language *)

BEGIN
  i := 255;
  j := 33;
  WRITELN (I,J,' AND = ',Iand (I, J));
  END.
```

The Assembler file is:

```
        .FUNC   IAND,2                  ; two arguments
        RORG    0
        MOVE.L  (A7)+,A0                ; return address
        MOVE.W  (A7)+,D0                ; J
        MOVE.W  (A7)+,D1                ; I
        AND.W   D1,D0                   ; I AND J
        MOVE.W  D0,(A7)
        JMP     (A0)
        .END
```

In the example given above we have made little attempt to make the assembly language procedure mimic the structure of a procedure generated by the Pascal Compiler. A complete description of this structure requires some preliminary discourse.

Automatic stack expansion code makes procedure entries a little complicated. To ensure that the stack segment is large enough before the procedure is entered, the compiler emits code to 'touch' the lowest point that will be needed by the procedure. If we 'touch' an illegal location (outside the current stack bounds), the MMU hardware signals a bus error which causes the 68000 to generate a hardware exception and pass control to an exception handler. This code, provided by the operating system, must be able to restore the state of the world at the time of the exception, and then allocate enough extra memory to the stack that the original instruction can be re-executed without problem. To be able to back up, the instruction that caused the exception must not change the registers, so a TST.W instruction with indirect addressing is used.

"LisaPascalDevManFeb82_027.PICT" 65 KB 2000-12-12 dpi: 72h x 72v pix: 1320h x 1977v

In the normal case, the procedure's LINK instruction should be preceded by a TST.W e(A7) which attempts to reach the stack location that can accomodate the static and dynamic stack requirements of the procedure. If the static and dynamic stack requirements of your assembly language procedurez are less than 256 bytes, you can assume that the compiler's fudge factor will protect the assembly language procedure, so the TST.W can be omitted. If the requirements are greater than 32K bytes, e(A7) may not be sufficient because only 16 bits of addressability are available (the 68000 does call a 16-bit processor). In this case, the compiler currently emits code something like:

```
        MOVE.L  A7,A0
        SUB.L   #Size,A0        ;#size=dynamic + static requirements
        TST.W   (A0)
```

If the compiler option D+ is in effect (the default), the first eight bytes of the data area following the final RTS or JMP (A0) contain the procedure name. LisaBug gets the procedure name from this block, making debugging much more pleasant. The following example is provided to show how an assembly language programmer can provide LisaBug with all the information it needs to perform fully symbolic low level debugging.

```
;
;   ASSEMBLY LANGUAGE EXAMPLE
;
DEBUGF  .EQU    1               ; true => allow debugging with proc names

;
;   HEAD -- This MACRO can be used to signal the beginning of an assembly
;   language procedure.  HEAD should be used when you do not want to
;   build a stack frame based on A6, but do want debugging information.
;
;   No arguments

        .MACRO  HEAD
        .IF     DEBUGF
        LINK    A6,#0           ; fancy NOP just for debugging purposes
        MOVE.L  (A7)+,A6
        .ENDC
        .ENDM

;
;   TAIL -- This MACRO can be used as a generalized exit sequence.  There
;   are two cases.  First, if you build a stack frame, TAIL can be used
;   to undo the stack frame, delete the parameters (if any) and return.
;   Second, if you do not want to build a stack frame based on A6,
;   this MACRO can be used to signal the end of an assembly language
;   procedure.  In either case if DEBUGF is true, the Procedure_name
;   is dropped by the MACRO as an 8 character name.
;
```


"LisaPascalDevManFeb82_028.PICT" 87 KB 2000-12-12 dpi: 72h x 72v pix: 1299h x 1989v

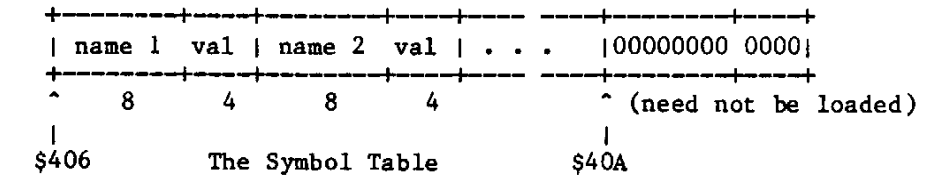
```
;   Three arguments:
;
;         1) Number of bytes of parameters to delete
;         2) Procedure_Name (maximum of 8 chars, no trailing blanks)
;         3) Number of blanks needed to complete 8 character name
;
;   If the 7.0 assembler is used the .BLOCK kludge can be removed.
;   The previous assembler (good old TLA) removes trailing blanks from
;   strings passed as arguments to a MACRO
;
        .MACRO  TAIL
        UNLK    A6
        .IF     %1 = 0
          RTS                           ; 0 bytes of parameters
        .ELSE
          .IF   %1 = 4
            MOVE    (A7)+,(A7)          ; 4 bytes of parameters
            RTS
          .ELSE
            MOVE    (A7)+,(A0)          ; 2 or >4 bytes of parameters
            ADD.W   #%1,A7              ; asm ADDQ.W or ADDA.W
            JMP     (A0)
          .ENDC
        .ENDC
        .IF     DEBUGF
        .ASCII  %2
        .BLOCK  %3,$20
        .ENDC
        .ENDM
;
;   The following example demonstrates the use of the TAIL macro for the
;   purpose of debugging.  The example assumes that you want to build
;   a stack frame based on A6.  In a real assembly language procedure the
;   zeroes below would be replaced by the local size and parameter size.

        .PROC   SIMPLE,0
        LINK    A6,#0           ; zero here indicates zero bytes of locals
        NOP                     ; body of procedure
        TAIL    0,'SIMPLE',2    ; zero here indicates zero bytes of parameters

        .END
```

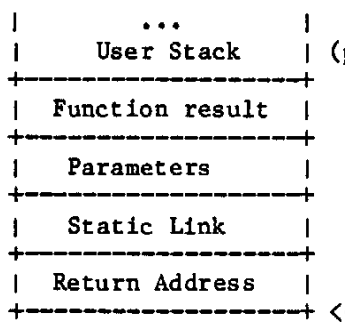

"LisaPascalDevManFeb82_029.PICT" 51 KB 2000-12-12 dpi: 72h x 72v pix: 1310h x 1983v

These macros are sufficient for the programmer writing small assembly language routines to be called from Pascal. If, however, you want the debugger to be able to handle symbols in a single huge .PROC (an interpreter, or an operating system, for example), you need to set up the symbol table yourself. The Assembler can create a .SYMBOLS file for you. Each entry in the file (and in the symbol table) contains 12 bytes. The first 8 bytes are the symbol name, left justified and padded with trailing spaces. The last 4 bytes are the symbol's value. The debugger uses location $406 to find the first entry in the table. Location $40A points to the first free entry (just past the last entry in the table).



To load your symbol table, load the .SYMBOLS file into memory, offset each of the symbol values by the loading address of your program, and reset the pointers in $406 and $40A to point to the new location of the table. The debugging version of the Monitor loads M.SYMBOLS into memory in exactly this manner so that the debugger can provide symbolic disassembly of the Monitor. If the symbols for the registers (RD0..RD7, and so on) are not in the table, the debugger appends them to the table when it is first invoked.

Upon entry to the assembly routine, the stack is:



D0-D2/A0-A1:	Scratch registers (can be clobbered)
D3,A2	Scratch registers, but should be preserved
D4-D7/A3,A4	Used for code optimization
A5:	Pointer to user globals (must be preserved)
A6:	Pointer to base of stack (must be preserved)
SP:	Top of stack

Registers D3 and A2 may be used at some time in the future by the compiler for code optimization, so the assembly language programmer should preserve them also.

The function result is present only if the Pascal declaration is for

"LisaPascalDevManFeb82_030.PICT" 88 KB 2000-12-12 dpi: 72h x 72v pix: 1317h x 1986v

a function. It is either one or two words. If the result fits in a single byte (a boolean, for example), the most significant half (the lower addressed half) gets the result value.

Parameters are present only if there are parameters. They are pushed on the stack in the order of declaration. All reference parameters are represented as 32 bit addresses. Value parameters less than 16 bits in size always occupy a full word. All non-set value parameters larger than 4 bytes are passed by reference. It is the procedure's responsibility to copy them. All large set value parameters are pushed onto the stack by the calling routine.

The static link is present only if the external procedure's level of declaration is not global. The link is a 4 byte pointer to the enclosing static scope.

It is the responsibility of the assembly language procedure to deallocate the return address, the static link (if any), and the parameters (if any). The SP must point to the function result or to the previous top of the stack upon return. Registers D4 through D7 and A3 through A7 must be preserved. It is recommended that you also preserve D3 and A2.

SPACE ALLOCATION DIRECTIVES

The space allocation directives are .ASCII, .BYTE, .WORD, .LONG, and .BLOCK.

```
        .ASCII  'string'
```

converts 'string' into the equivalent ASCII byte constants and places the bytes in the code stream. The string delimiters must be single quotes. To insert a single quote into the code:

```
        .ASCII  'AB'
        .BYTE   $27     ;ASCII equivalent of single quote
        .ASCII  'CD'
```

assembles the string AB'CD.

```
        .BYTE   <values>
```

allocates a byte of space in the code stream for each of the values given. Each value must be between -128 and 255.

```
        .BLOCK  <length>[,value]
```

allocates <length> bytes for each value listed. If no value is given, a block of zeros is allocated.

"LisaPascalDevManFeb82_031.PICT" 75 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1991v

```
        .WORD   <values>
```

allocates a word of space in the code stream for each of the values listed. The values must be between -32768 and 65535. For example,

```
TEMP    .WORD   0,65535,-2,17
```

creates the assembled output:

```
        0000
        FFFF
        FFFE
        0011
```

```
        .LONG   <values>
```

allocates two words of space for each value in the list. For example,

```
STUFF   .LONG   0,65535,-2,17
```

creates the output:

```
        00000000
        0000FFFF
        FFFFFFFE
        00000011
```

```
<label>  .EQU   <value>
```

assigns <value> to <label>. <value> can be an expression containing other labels.

```
        .ORG    <value>
```

puts the next byte of code at <value> relative to the beginning of the assembly file. Bytes of zero are inserted from the current location to <value>.

```
        .RORG
```

is similar to .ORG. It indicates that the code is relocatable. Because the loader does not support the .ABSOLUTE pseudo-op, RORG is mostly cosmetic.

RORG (without the leading period) is the same as .RORG. Similarly, END = .END, EQU = .EQU, PAGE = .PAGE, LIST = .LIST, NOL = .NOLIST, and TTL = .TITLE. The TLA directives .INTERP and .ABSOLUTE have not been implemented. The TLA directives .PRIVATE, .PUBLIC, and .CONST are currently unimplemented.

"LisaPascalDevManFeb82_032.PICT" 53 KB 2000-12-12 dpi: 72h x 72v pix: 1303h x 1993v

MACRO DIRECTIVES

A macro consists of a macro name, optional arguments, and a macro body. When the assembler encounters the macro name, it substitutes the macro body for the macro name in the assembly text. Wherever %n occurs in the macro body (where n is a single decimal digit), the text of the n-th parameter is substituted. If parameters are omitted, a null string is used in the macro expansion. A macro can invoke other macros up to five levels deep. In the assembly listing, macros are shown fully expanded and marked with a '#' in the left margin.

```
.MACRO  <identifier>
.
.
.ENDM
```

defines the macro named <identifier>. The macros HEAD and TAIL are defined above. As a further example, consider:

```
.MACRO Help
MOVE    %1,D0
ADD     D0,%2
.ENDM
```

If 'Help' is called in an assembly with the parameters 'Alpha' and 'Beta', the listing created would be:

```
        Help    Alpha,Beta
#       MOVE    Alpha,D0
#       ADD     D0,Beta
```


"LisaPascalDevManFeb82_033.PICT" 44 KB 2000-12-12 dpi: 72h x 72v pix: 1302h x 1974v

CONDITIONAL ASSEMBLY DIRECTIVES

The conditional assembly directives .IF, .ELSE, and .ENDC are used to include or exclude sections of code at assembly time based on the value of some expression.

```
.IF     <expression>
```

identifies the beginning of a conditional block. <expression> is considered to be false if it evaluates to zero. Any non-zero value is considered true. The expression can also involve a test for equality (using <> or =). Strings and arithmetic expressions can be compared. If <expression> is false, the Assembler ignores code until a .ELSE or .ENDC is found. The code between the optional .ELSE and .ENDC is assembled if <expression> is false. Otherwise it is ignored. Conditionals can be nested. The macros HEAD and TAIL given above provide examples of the use of conditionals. The general form is:

```
.IF     <expression>
.               ;assembled only if <expression> is true
.
[.ELSE]         ;optional
.               ;assembled only if <expression> is false
.
.ENDC
```

EXTERNAL REFERENCE DIRECTIVES (.REF and .DEF)

Separate routines can share data structures and subroutines by linkage between assembly routines using .DEF and .REF. These directives cause the Assembler to generate link information that allows separately compiled assembly routines to be linked together. .DEF and .REF associate labels between assembly routines, not between assembly routines and Pascal. The Linker resolves the references.

```
.DEF    <identifier-list>
```

identifies labels defined in the current routine as available to other assembly routines through matching .REFs. The .PROC and .FUNC directives also generate a .DEF with the same name, so assembly routines can call external .PROCs and .FUNCs with .REFs.

"LisaPascalDevManFeb82_034.PICT" 71 KB 2000-12-12 dpi: 72h x 72v pix: 1314h x 1986v

```
        .PROC Simple,1
        .DEF Alpha, Beta
        .
        .
        BNE     Beta
        .
        .
Alpha   MOVE
        .
        RTS
Beta    MOVE
        .
        RTS
        .END
```

This example defines two labels, Alpha and Beta, which another assembly routine can access with .REF.

```
        .REF    <identifier-list>
```

identifies the labels in <identifier-list> used in the current routine as available from some other assembly routines which used .DEFs.

```
        .PROC Simple
        .REF Alpha
        .
        .
        JSR     Alpha
        .
        .END
```

uses the label 'Alpha' declared in the .DEF example.

When a .REF is encountered, the assembler generates a short absolute addressing mode for the instruction (the opcode followed by a word of 0's). The assembler's second pass transforms each of these into a short external reference with an address pointer to the word of 0's following the opcode. If the referenced label and the reference are in the same segment module, the Linker changes the addressing mode from short absolute to single word PC relative. If, however, the referenced procedure is in a different segment, the Linker converts the reference to an indexed addressing mode (off A5) and the word of zeros is converted into the proper entry offset in the jump table. If the referenced procedure is in an intrinsic unit (and therefore in a different segment), the IUJSR, IULEA, IUJMP, and IUPEA instructions are used (see page ##). The Linker blindly assumes that the word immediately before the word of zeros is an opcode in which the low order 6 bits are the effective address. Thus, a .REF label cannot be used with any arbitrary instruction. The .REF labels are intended for JSR, JMP, PEA, and LEA instructions.

"LisaPascalDevManFeb82_035.PICT" 67 KB 2000-12-12 dpi: 72h x 72v pix: 1316h x 1986v

LISTING CONTROL DIRECTIVES

The directives that control the Assembler's listing file output are .LIST, .NOLIST, .PAGE, and .TITLE. If you do not specify a name for the listing file in response to the Assembler's prompt:

```
Listing file (<cr> for none) -
```

the listing directives are ignored.

.LIST and .NOLIST

can be used to select portions of the source to be listed. The listing goes to the specified output file when .LIST is encountered. .NOLIST turns off the listing. .LIST and .NOLIST can occur any number of times during an assembly.

.PAGE

inserts a page feed into the listing file.

.TITLE '<title>'

specifies a title for the listing page. <title> can contain up to 80 characters.

```
.TITLE 'Interpreter'
```

places the word, Interpreter, at the head of each page of the listing.

FILE DIRECTIVE

The pseudo-op

```
.INCLUDE <filename>
```

causes the contents of <filename> to be assembled at the point of the .INCLUDE. <filename> need not specify the .TEXT suffix. The last character of the filename must be the last non-space character on the line--do not put a comment on this line. An included file cannot itself contain a .INCLUDE statement.

"LisaPascalDevManFeb82_036.PICT" 52 KB 2000-12-12 dpi: 72h x 72v pix: 1301h x 1985v

LISABUG

LisaBug allows you to examine, disassemble, and change the contents of memory, set breakpoints, and do immediate assemblies. If the compiler D option is on (the default), procedure names are available to the debugger, and Lisabug uses the symbols wherever appropriate.

Type M to the Monitor command prompt to invoke LisaBug. It asks:

```
What file?
```

You can type <cr> to enter LisaBug without any file. If you type a file name, that code file is loaded into LisaBug. The LisaBug command prompt is '>'. The default radix is hexadecimal.

You can drop into LisaBug by hitting the NMI key which is currently the third key from the left in the top row of the numeric keypad.

A FEW EXAMPLES

If you type a file name, LisaBug starts up with the program counter at the start of the program. To see one instruction disassembled (say at 32F96), type

```
>ID 32F96
```

(followed by RETURN, of course). ID stands for Immediate Disassemble. Each subsequent ID command, if given without any address, disassembles the next instruction found. In addition to printing the value of each byte, LisaBug prints the ASCII equivalent of that value, if a printable one exists. If none exists, it prints a period.

To disassemble 20 consecutive addresses, type

```
>IL
```

IL (Immediate Disassemble Lines) can also be followed by an address. Subsequent IL commands disassemble successive blocks of 20 consecutive locations in memory.

If the object file being examined was compiled with the D+ compiler option, the procedure names are available in LisaBug and can be used in any expressions. For example,

```
>IL Foo 5
```

disassembles the first 5 lines of procedure 'Foo'.

```
>BR Foo+40
```

sets a break point 40 bytes into procedure 'Foo'.

"LisaPascalDevManFeb82_037.PICT" 72 KB 2000-12-12 dpi: 72h x 72v pix: 1302h x 1987v

You can also use labels in immediate assemblies:

```
>sy Ken 6000
>A Ken NOP
```

assembles a NOP instruction at the address 'Ken'.

```
>A 6000 <cr>
>Rich: TAS $100
> <cr>
```

enters the immediate assembler at 6000, defines the label 'Rich', and assembles a TAS instruction.

THE SYMBOL TABLE

The symbol table is the union of the user symbol table and the distributed procedure names. The user symbol table contains the user declared symbols (like 'KEN' in the example above) and the predefined symbols (RD0 and friends). Each entry contains twelve bytes. The first eight bytes are the symbol name, and the last four bytes are the symbol's value. Location $406 gives the beginning of the symbol table, and $40A points to the end of the table. The section 'Communication with Pascal' in the Assembler chapter of this manual contains more information about the symbol table.

"LisaPascalDevManFeb82_038.PICT" 43 KB 2000-12-12 dpi: 72h x 72v pix: 1310h x 2002v

LISABUG COMMANDS

Definitions:

Constant	A constant in the default base
$Constant	A hex constant
&Constant	A decimal constant
'ASCII String'	An ASCII string
Name	A symbol in the symbol table
RegName	RD0..RD7, RA0..RA7, PC, US, or SS. A predefined symbol in the symbol table with a value set by LisaBug. The value is equal to the value of the register in question. LisaBug automatically updates the values of these symbols. The 'R' is appended to distinguish the register names from hexadecimal numbers.
Expr	An expression. Expressions can contain names, regnames, strings, and constants. Legal operators are + - * /. Expressions are evaluated left to right. * and / take precedence over + and -. (and) can be used to indicate indirection. < and > can be used to nest expressions. In those cases where an odd value is probably a mistake, LisaBug warns you that you are trying to use an odd address. If you decide to go ahead, it subtracts one from the address given. If the compiler option D+ is used, procedure names are also legal in expressions.
Exprlist	A list of expressions separated by blanks.
Register	D0..D7, A0..A7, PC, SR, US, or SS. Note that A7 is SP (the stack pointer).

Moving the LisaBug Window:

P expr — Set port number to expr. Valid port numbers are:

0	Lisa keyboard and screen (default)
1	UART Port A (farthest from Power Supply)
2	UART Port B

If you move the port to a UART, you must have a modem eliminator connected to that port.

"LisaPascalDevManFeb82_039.PICT" 64 KB 2000-12-12 dpi: 72h x 72v pix: 1350h x 1988v

Symbols and Base Conversion:

SY	Display the values of all symbols
SY name	Display the value of the symbol name
SY name expr	Assign expr to the symbol name
CV exprlist	Display the value of each expression in hex and decimal.
SH	Set the default radix to hex
SD	Set the default radix to decimal

Assembly and Disassembly:

A expr statement

A expr

Assemble one statement (instruction) at expr. If you use the form A expr, LisaBug asks you for the statement to be assembled. You can continue assembling instructions into consecutive locations. Type <cr> to exit the immediate assembler.

ID	Disassemble one line at the next address
ID expr	Disassemble one line at expr
IL	Disassemble 20 lines at the next address
IL expr	Disassemble 20 lines starting at expr
IL expr1 expr2	Disassemble expr2 lines starting at expr1

Upon entering LisaBug, the 'next address' is the current PC.

"LisaPascalDevManFeb82_040.PICT" 42 KB 2000-12-12 dpi: 72h x 72v pix: 1332h x 1978v

Set, Display, and Find Memory:

SM exprl exprlist
Set memory with exprlist starting at exprl. SM assumes that each element of exprlist is 32 bits long. To load different length quantities, use SB or SW described below. If the expression given is longer than 32 bits, SM takes just the upper 32. For example, if we ask LisaBug to:

```
SM 1000 'ABCDE'
```

it deposits the ASCII equivalent of 'ABCD' starting at 1000.

SB exprl exprlist
Set memory in bytes with exprlist starting at exprl

SW exprl exprlist
Set memory in words with exprlist starting at exprl

SL exprl exprlist
Set memory in long words with exprlist starting at exprl. For example,

```
SL 100 1
```

is equivalent to

```
SM 100 0000 0001
```

DM expr — Display memory at expr. DM RA3+10, for example, displays the contents of memory from the address pointed to by A3 for 10 bytes. DM (110) displays the contents of the memory location addressed by the contents of location 110.

DM exprl expr2 — Display memory. If exprl < expr2, then display memory from exprl to expr2. Otherwise, display memory for expr2 bytes starting at exprl.

DB expr — Display memory as bytes.

DW expr — Display memory as words.

DL expr — Display memory as long words.

FB starting_addr count data — Find Byte.
Find the byte or bytes 'data' in memory between 'starting_addr' and 'starting_Addr'+'count'.

FM starting_addr count data — Find Memory

FW starting_addr count data — Find Word

"LisaPascalDevManFeb82_041.PICT" 62 KB 2000-12-12 dpi: 72h x 72v pix: 1347h x 1974v

FL starting_addr count data Find Long

Set and Display Registers:

TD Display the Trace Display at the current PC

register Display the current value of the register. D0, for example, is a command to LisaBug to display the current value in the register D0. RD0, on the other hand, is a name automatically placed in the symbol table to give you a handle on the contents of D0 in an expression.

register expr Set the register to expr

Memory Management:

LP expr Convert logical address to physical address.

DO expr Set the SEG1/SEG2 bits. These bits determine the hardware domain number. If the Status Register shows that you are in supervisor state, then the effective domain is zero, and the domain number returned by LisaBug is the domain which would be active if the SR were changed to user state.

WP 0 or 1 Diable (0) or Enable (1) Write Protection. The default is 1.

MM start [end_or_count]
MM with one or two arguments displays information about the MMU registers. The second argument defaults to 1. If the starting address is greater than the second argument, the second argument is a count of the number of MMU registers to be displayed. If the starting address is less than the second argument, the second argument is the last register displayed.

```
MM 70
```

displays

```
Segment[70] Origin[000] Limit[00] Control[C]
```

These values are the Segment Origin, Limit, and Control bits stored by the hardware for each MMU register. As can be seen from a careful perusal of the hardware documentation, a Control value of C means the segment in question is unused (invalid). If the Control value is valid (F, for example), the debugger also displays the Physical Start

"LisaPascalDevManFeb82_042.PICT" 71 KB 2000-12-12 dpi: 72h x 72v pix: 1321h x 1989v

and Stop addresses of the segment.

```
MM &100 8
```

displays the MMU register information for the 8 registers starting at register 64 (decimal 100).

MM num org lim cntrl

The MM command followed by four arguments sets the MMU information for segment 'num'. The Origin, Limit, and control bits can be changed. The Monitor uses the first 16 registers, so it is safer not to mess with them.

```
MM 70 100 ff 7
```

sets the Origin of segment 70 to 100 and the control bits to 7 (a regular segment). The segment limit of -1 makes the segment 512 bytes long.

Breakpoints, Patchpoints, Traces, Calls:

BR — Display the breakpoints currently set. Up to 16 breakpoints can be handled by LisaBug. Break points are displayed both as addresses and as symbols. An asterisk marks the point of the breakpoint in the disassembly. Patch points are marked with '!'.

BR exprlist — Set each breakpoint in exprlist. Symbols are legal, of course, so we can:

```
BR Ralph+4
```

if Ralph is a known symbol.

PA insertion_addr destination_addr

Insert a Patch. PA can be used to insert a sequence of code terminated by a TRAP #$f into another sequence of code. Lisabug maintains a table of patches and return addresses to implement this facility. The trace command works with patches. It displays the next instruction to be executed and its environment. You can have up to 16 patches. A patch can be removed by using the CL (Clear) command with the patch insertion address.

PA — Display patch addresses

CL — Clear all breakpoints and patchpoints

CL exprlist — Clear each breakpoint or patchpoint in exprlist

"LisaPascalDevManFeb82_043.PICT" 67 KB 2000-12-14 dpi: 72h x 72v pix: 1304h x 1938v

G	Start running at the current PC
G expr	Starting running at expr
T	Trace one instruction at the current PC
T expr	Trace one instruction at expr
CA expr	Call a subroutine in the debugger's environment.
SC expr	Stack Crawl. Display the user call chain. Expr sets the depth of the display. It can be omitted.
Q	Exit LisaBug, if it was called from Talk
RM	Return to the Monitor. RM checks the interrupt level, stops exec files, and sets the domain to zero. If you are in an interrupt handler, RM may refuse to do anything. If the SR shows 2nxx where n is not zero, you are in an interrupt handler. To get back to the monitor, type G, hit NMI, and try RM again. With any luck, you will escape eventually.
RB	Reboot. The Lisa is reset. Reboot the Apple II and the Lisa should also reboot automatically.

Overcome Inadequate Hardware:

DU expr	Disk unclamp. The Twiggy may not reliably eject the disk at the right time. If you have trouble, try the DU command followed by the drive number. Valid drive numbers are 1 and 2.
DC expr	Disk Clamp. If the Twiggy refuses to suck up the disk and clamp it in place, try the DC command followed by the drive number (1 or 2).
RS	Display the patch Return address Stack

If you have the debug card,

DR	Display index or ranges of dump RAM.
MR	Set a value level #5 interrupt on a word change.

"LisaPascalDevManFeb82_044.PICT" 60 KB 2000-12-12 dpi: 72h x 72v pix: 1316h x 1991v

```
register            Display the current value of the register.
register expr       Set the register to expr
A  expr statement
A  expr             Assemble one statement (instruction) at expr.
BR                  Display the breakpoints currently set.
BR exprlist         Set each breakpoint in exprlist.
CA expr             Call a LisaBug subroutine
CL                  Clear all breakpoints and patchpoints
CL exprlist         Clear each breakpoint or patchpoint in exprlist
CV exprlist         Display the value of each expression in hex and decimal.
DB expr             Display memory as bytes.
DC expr             Disk Clamp.
DL expr             Display memory as long words.
DM expr1 expr2      Display memory.
DO expr             Set the SEG1/SEG2 bits.
DR                  Display index or ranges of dump RAM.
DU expr             Disk unclamp.
DW expr             Display memory as words.
FB starting_addr count data            Find Byte.
FL starting_addr count data            Find Long
FM starting_addr count data            Find Memory
FW starting_addr count data            Find Word
G                   Start running at the current PC
G expr              Starting running at expr
ID                  Disassemble one line at the next address
ID expr             Disassemble one line at expr
IL                  Disassemble 20 lines at the next address
IL expr             Disassemble 20 lines starting at expr
IL expr1 expr2      Disassemble expr2 lines starting at expr1
LP expr             Convert logical address to physical address.
MM expr1 expr2      Display MMU information
MM num org lim ctrl Set MMU information
MR                  Set a value level #5 interrupt on a word change.
P  expr             Set port number to expr.
PA insrtaddr destaddr Insert a Patch.
PA                  Display patch addresses
Q                   Exit LisaBug, if it was called from Talk
RB                  Reboot.
RM                  Return to the Monitor.
RS                  Display the patch Return address Stack
SB expr1 exprlist   Set memory in bytes with exprlist starting at expr1
SC expr             Stack Crawl.
SD                  Set the default radix to decimal
SH                  Set the default radix to hex
SL expr1 exprlist   Set memory in long words with exprlist starting at expr1.
SM expr1 exprlist   Set memory with exprlist starting at expr1.
SW expr1 exprlist   Set memory in words with exprlist starting at expr1
SY                  Display the values of all symbols
SY name             Display the value of the symbol name
SY name expr        Assign expr to the symbol name
T                   Trace one instruction at the current PC
T  expr             Trace one instruction at expr
TD                  Display the Trace Display at the current PC
WP 0 or 1           Diable (0) or Enable (1) Write Protection.
```


"LisaPascalDevManFeb82_045.PICT" 93 KB 2000-12-12 dpi: 72h x 72v pix: 1365h x 2052v

INTRINSIC UNIT INSTRUCTIONS

LisaBug recognizes the Intrinsic Unit Instructions IUJSR, IUJMP, IULEA, and IUPEA. These instructions use the Line 1010 exception handler, and are created by the Linker. Full details can be found in the Development System Internal Documentation.

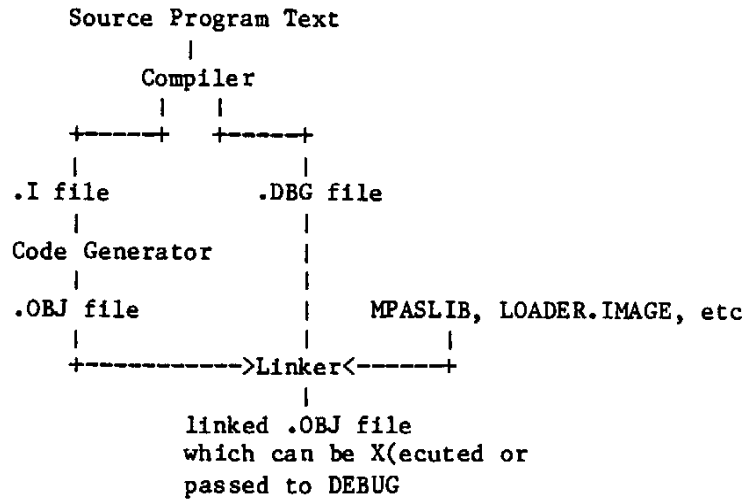
"LisaPascalDevManFeb82_046.PICT" 22 KB 2000-12-12 dpi: 72h x 72v pix: 1320h x 2001v

THE SYMBOLIC DEBUGGER

File Needed: DEBUGGER.OBJ
(the debugger is currently broken)

The debugger allows the programmer to follow the execution of a Pascal program, set traces and break points, and display or change variables without the need to insert cumbersome diagnostic statements. The basic unit of program execution is the Pascal statement and the basic unit of program data is the variable. If the debugger is present, run time errors drop the program into the debugger.

To use the debugger, first create a .DBG file in the compiler, then link the .DBG file with the output of the code generator, MPASLIB, and any other units that are needed. The .OBJ file that results can be executed normally, or you can X(ecute DEBUGGER and hand it the .OBJ file. The D(ebug option in the Monitor command line also invokes the Debugger.



Once executed, DEBUG takes the .OBJ file and asks you for a command. The first "command:" prompt occurs just after the start up of the program being debugged. The available commands are:

Set <proc [:offset] | address>

Set a breakpoint at the location given. PROC is the name of a procedure or function. The offset, if given, specifies the byte offset from the beginning of the procedure within the object code of the procedure. You can disassemble the procedure code and compare it with the source to decide where to set the breakpoint. ADDRESS is the absolute location at which the breakpoint is set. All numbers in the debugger are in decimal radix unless prefixed with '$' (for hexadecimal). The address given

"LisaPascalDevManFeb82_047.PICT" 75 KB 2000-12-12 dpi: 72h x 72v pix: 1300h x 1996v

must be in core and within the code image.

Set Value <VarID | Address [size]>

Set Value sets a value breakpoint at the address given, or at the address of the variable identifier given. VarID can be either an entire variable, or just a field of it. When a check point is encountered therafter, the debugger checks all the value breaks. If the value of the variable or address has changed, the debugger stops, prints the old and new values of the variable or address in question, and waits for the next command. If you Set Value p^ (which sets a value breakpoint on the variable pointed to by p), do not change the value of p. You cannot set a value breakpoint on a field of a packed variable.

Set <VarID>

Deposit a value in VarID. The debugger waits for the value to be typed. To retain the old value, type <cr>. To set the value of a string variable, just type the new value (without quotes). It is not possible to set a string variable to the null string. String variables cannot be subscripted.

Set <Reg> <Num>

Deposit a value into one of the 18 registers (RD0-RD7, RA0-RA7, RPC, and RSR).

Set ..

Set "array abbreviation" mode. While in this mode, the present values in any array that is not a packed array of char are presented as first value, second value,"..", last value.

Clear ..

Clear .. gets you out of "array abbreviation" mode.

Clear

Clear clears all breakpoint and trace settings.
You cannot clear the breakpoint which caused you to drop into the debugger.

Clear <Proc [:offset]>

Clears the breakpoint set by Set <Proc [:offset]>.

Clear Value

Clear Value clears all value breakpoints and traces.

Clear Value <VarID | Address [Size]>

"LisaPascalDevManFeb82_048.PICT" 71 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1993v

Clear Value removes the breakpoint specified.

List Break

List Break lists the current break and trace points.

List <Proc [:offset] | Address>

List disassembles 20 lines of code starting at the address given. Sometimes fewer than 20 lines are disassembled. To continue the disassembly from this point, type List (without any arguments).

Regs

Regs lists the current values of all the user registers in hexadecimal.

Trace

Trace puts a trace on all procedure entries, but does not display procedure parameters.

Trace Proc

Trace Proc traces all procedure and function entries, displaying the parameters.

Trace Proc <Proc [:offset] | Address>

When an argument is given to Trace Proc, a trace is set in that procedure at the offset given. The parameters to the procedure are also displayed.

Trace <Proc [:offset] | Address> also puts a trace on the address given. The parameters to the procedure, howver, are not displayed.

Trace Value <VarID>

Trace value reports changes in the value of <VarID>, but does not stop execution.

Crawl

Crawl allows you to crawl up the stack as far as you like. Crawl starts at the current PC. At each stack location Crawl prints the current value, then waits for a carriage return. Any response other than <cr> exits Crawl and returns to the command request of the Debugger.

MacsBug

The Macsbug command drops you into Lisabug (the low level debugger). When entered in this manner, you are still in the debugger's environment. To get into Lisabug in the program's environment, first get the current PC of the program (from the Debugger), set a

"LisaPascalDevManFeb82_049.PICT" 67 KB 2000-12-12 dpi: 72h x 72v pix: 1321h x 1992v

Lisabug breakpoint at that location (BR or G TILL), then return to the debugger. Give the debugger the GO command, and when the program is started back up, you are dropped into Lisabug in the correct environment.

Check <Proc [:offset] | Address>

The debugger needs to be told when to check value breakpoints and traces. Check sets the location at which you want these checks to be made.

Go

Go restarts the program being debugged.

Quit

Quit causes the debugger to return you to the Monitor.

You can also poke around inside your code with the debugger. If you respond to the Command request with a name or a number, DEBUG responds as follows:

Name's Type	Action
Procedure or Function	Display segment number and offset within segment of that procedure or function, and the routine's in-code address, if any.
Variable identifier	Display the current value of the variable. The variable can (but need not) be qualified by field selection, indexing, or subscripting. Subscripts can be either integer constants, or a variable of the appropriate type.
Number	Convert hex to decimal, or vice versa, and display 16 bytes starting at that memory location. If there is an equivalent <Proc [:offset]> value, that too is reported.
<Number> TYPEID	Print memory starting at <number> as if it were a variable of type TYPEID.

You should not use any identifiers in your program that conflict with DEBUG command names. If your program terminates normally during debugging, DEBUG also quits.

If DEBUG does not understand a command, it merely asks you again for a command. In general, if DEBUG does not say it has done something, then it has taken no action at all.

To interrupt a program and drop into the debugger at almost any point, hit the NMI button on the side of the machine. In certain situations NMI may drop you into LisaBug instead of the debugger.

"LisaPascalDevManFeb82_050.PICT" 78 KB 2000-12-12 dpi: 72h x 72v pix: 1322h x 1997v

At link time, remember to include the .DBG extension when loading the debug file into the .OBJ file. The .DBG file must follow the .OBJ file to which it corresponds in the link. All separate compilation units must precede the main program in the link. If you want to use the .DBG files, you cannot do a partial link. The .DBG files and the resulting debuggable .OBJ files are significantly larger than the corresponding .OBJ files. The Debugger takes up about 10K of RAM plus about 60 words per procedure.

"LisaPascalDevManFeb82_051.PICT" 31 KB 2000-12-12 dpi: 72h x 72v pix: 1305h x 2000v

THE FILER

File Needed: FILER.OBJ

INTRODUCTION

The Filer is modeled after the UCSD P-system's Filer and provides a similar set of functions. However, there are some small but important differences between the two.

The Monitor Filer requires that a colon follow a volume name in every case. It provides access to as many as 20 on-line volumes. The maximum number of files in a volume directory is 77.

All "workfile" commands and workfile-oriented features of the UCSD Filer have been omitted from the Monitor Filer. The functions of the Monitor utility programs Flipdir and Verify are provided by the Filer commands "S(ex" and "V(erify," respectively.

The UCSD Filer's "V(olume" command has been changed to "O(n-line" in the Monitor Filer. The UCSD Filer's "eX(amine" command is not available in the Monitor Filer.

The Monitor Filer's "T(ransfer" command performs automatic verification of all transfers between blocked devices.

ESCAPE aborts the currently executing function. When a wildcard R(emove or C(hange is aborted, you are asked whether to update the directory. A response of ESCAPE to this question is interpreted as 'No'.

The Monitor Filer includes a volume manager subsystem that permits you to maintain and manage the volume population on a Corvus drive. This subsystem, accessible through the "M" command, replaces the old VMGR utility.

FILER COMMANDS

The following is a list of commands that are recognized by the Filer. Filer commands are invoked by pressing the key which corresponds to the first letter of the command name.

B(ad-blocks - Scans for and reports bad blocks on blocked device.

C(hange - Changes a volume or file name on a blocked device. "Wildcard" file name specifications are recognized.

D(ate - Sets or changes the system date.

E(xtended directory listing - Provides a detailed list of the contents of a blocked volume. "Wildcard" file names specify the display of a directory subset. You can write a directory to a printer with E #4:,PRINTER:

"LisaPascalDevManFeb82_053.PICT" 82 KB 2000-12-12 dpi: 72h x 72v pix: 1305h x 1993v

K(runch - Creates the largest possible block(s) of contiguous space on a blocked volume by relocating existing files on that volume. It's a good idea to scan a volume for bad blocks before any attempts are made to Krunch it. Do not Krunch a volume that has bad blocks.

L(ist directory - Provides an summary of the contents of a blocked volume. See "E(xtended directory listing," above.

N(ew - Creates a directory entry with the specified file name. Any volume name used to prefix the file name must be that of an on-line, blocked device. You can attach a size-specification suffix to the end of the file name. This suffix indicates the number of blocks to be occupied by the new file. The suffix consists of a non-negative integer constant or an asterisk ("*"), enclosed in square brackets ("[]"). For example,

```
FARLEY:MYFILE.TEXT[40]
XRAY.OBJ[*]
```

The new file is placed on the specified volume in the first empty space that is large enough to hold it. The asterisk indicates that the file should fill half the largest free area on the volume, or all of the second-largest area, whichever is larger. In the absence of a size specification, the newly-created file occupies the largest area of contiguous free blocks on the volume. Files created with N(ew are stamped with the current system date, while the storage areas to which they correspond are left unaltered. N(ew permits the creation of zero-length files.

O(n-line - Provides a list of all volumes that are on-line.

P(refix - Changes the system prefix volume name.

Q(uit - Exits the Filer.

R(emove - Deletes entries from the directory of a blocked volume on a single-file or "wildcard" basis.

S(ex - Performs sexual reassignment of a blocked volume's directory. This command corresponds to the FLIPDIR utility. The Lisa is a female machine, whereas the Apple II is a male machine.

T(ransfer - Copies and transfers information between volumes. Single-file or multiple-file wildcard transfers are allowed. You can also transfer between blocked and unblocked volumes. Transfers between blocked volumes are automatically verified, but transfers involving unblocked volumes are not.

"LisaPascalDevManFeb82_054.PICT" 87 KB 2000-12-12 dpi: 72h x 72v pix: 1324h x 1990v

Source: David T. Craig | Page 0057 of 0107

V(erify - Compares blocked files for equality. You can compare single-files or multiple files common to two blocked volumes. Wildcard specifications can be used to name the comparands, so subsets of the files on one volume can be compared with a congruent subset of files on another volume. Verify detects and reports the following situations:

* The "source" and "destination" files match;
* The source differs from the destination in date-stamp, size, and/or contents (contents are always compared if sizes match, whether or not dates match);
* No counterpart to a given source file exists on the destination volume.

The report produced by Verify can be redirected to a device or file other than the console by following the destination file/volume name specification with a comma, then the name of the desired output device or file. For example,

```
Verify what file/vol  ? VOL1:,VOL2:,PRINTER:
```

is equivalent to:

```
Verify what file/vol  ? VOL1:
Against what file/vol ? VOL2:,PRINTER:
```

The verification report in either of these cases is diverted to the PRINTER: device.

Z(ero - Erases and initializes the directory area of a blocked volume. If the volume already has a directory prior to the Z(ero, you have the option of retaining the old volume name and/or volume size. Z(ero can be used to increase or decrease the size of the virtual volume MEMORY:. Caution should be exercised, however, because it is possible to specify a volume size that is much larger than the LISA memory complement permits. In this case, a "memory overflow" is reported, and you should again invoke Z(ero to shrink MEMORY: to a reasonable size. Do not leave the Filer or attempt to use MEMORY: after receiving the "memory overflow" message!

Remember that Z(ero produces an empty directory. Therefore, to change the size of MEMORY: without erasing the directory, you must still use the CHANGEMEM utility.

vM(gr - Enters the volume manager (vMgr) subsystem, which presents its own sub-menu, and offers the following commands:

L(ist - List the hard disk Volumes (like Filer's O(n-line command). From time to time, you may destroy the directory of one or more volumes that reside on the hard disk. The vMgr

"LisaPascalDevManFeb82_055.PICT" 89 KB 2000-12-12 dpi: 72h x 72v pix: 1340h x 1991v

subsystem assigns temporary names to these "bad" volumes so that you can be warned of their contamination, and can also manipulate them, if necessary. The form of such temporary names is BAD*n, where n is an integer (e.g., BAD*1, BAD*10, etc). Temporary names for "bad" volumes are effective only within the vMgr subsystem.

M(ount - M(ount assigns a hard disk volume to a specific Monitor device number, taken from the set [4,5, 9..20]. You can specify the device number to which a volume is associated, or you can accept the default selected by the vMgr. When vMgr picks a default unit number, it chooses the highest number that is not currently in use.

N(ew - N(ew creates new volumes. You can accept the default values for volume size and location as offered by the vMgr, or specify your own.

Q(uit - Leave vMgr subsystem. If you have made any changes to the volume table you must confirm whether or not they should be made permanent in the default mount table. If you respond with any character other than 'Y', any changes made are temporary -- when the system is rebooted, the original settings will take effect.

R(emove - R(emove unmounts and destroys a volume. You can R(emove a M(ounted volume, but to do so you must approve the U)nmounting of that volume.

U(nmount - U(nmount is comparable to removing a floppy from a drive. It disassociates the volume from the unit on which it was mounted. The U(nmounted volume and the data it contains still exist on the hard disk drive, but can not be accessed through any Monitor device.

W(rite-protect - W(rite-protect toggles the write-protection status for a volume. The contents of a write-protected volume cannot be changed. This command changes the default mount table. Newly-created volumes are not write-protected.

See the Apple Pascal Operating System Reference Manual for further information.

Apple 2?

"LisaPascalDevManFeb82_056.PICT" 79 KB 2000-12-12 dpi: 72h x 72v pix: 1373h x 2004v

Source: David T. Craig | Page 0059 of 0107

PROFILE OR CORVUS INSTALLATION

To connect a Corvus or Profile hard disk to your Lisa:

1. Turn everything off. You need two floppy disk drives, a controller card, and the latest monitor release with both the male and female boot volumes.

   Set up the Apple so that drives #4 and #5 are available. Do not connect the hard disk yet.

2. Place the male boot volume in #4: and the female boot volume in #5:. Turn on the Apple, the Lisa, and the hard disk. After awhile, the Monitor command line should appear on the Lisa.

   Plug the hard disk cable into the parallel port on the back of the Lisa. If the monitor command line starts flickering, type RETURN.

3. Execute NEWZERO and respond 'Y'. NEWZERO initializes the hard disk mount and volume tables.

4. Execute the Filer and enter the volume Manager (type M).

5. Create (N(ew) a volume (MBOOT:) with 2048 blocks of space. Mount it as #20: (or anything other than #5:). Q(uit the Volume Manager and update the mount table (type 'Y').

6. Transfer all of the files on the female boot diskette (#5:) to the new volume (#20:). T #5:=,#20:$

7. Enter the Volume Manager again and mount the new volume as #5:. Q(uit the volume manager and update the mount table.

9. Turn everything off. Set up the floppy disk drives as #9: and #4: (if you have two disk controller cards). Place the male boot volume in #4:. Turn on the Apple, the Lisa, and the hard disk.

10. Copy the entire monitor release onto the female boot volume (#5:), but do not overwrite any files that are already on #5:.

"LisaPascalDevManFeb82_057.PICT" 67 KB 2000-12-12 dpi: 72h x 72v pix: 1315h x 1997v

THE EDITOR

File needed: EDITOR.OBJ
LISA:EDITOR.FONT
LISA:EDITOR.MENUS
LISA:SYSTEM.FONT

INTRODUCTION

The mouse oriented editor is invoked by the monitor command E. Unlike the UCSD editor (invoked by U), this editor adheres to the Lisa User Interface. When invoked, it displays its menu, a portion of the Scrap folder, and a dialog box which asks you for the name of the file to be edited:

Get Document named?

Type the name of the desired file, followed by <RETURN>. The editor opens a folder and displays the first portion of the file. To open an empty folder (to start a new file), type just <RETURN> to the request for a document name.

The arrow or I-beam shows the current position of the mouse. The blinking vertical bar marks the insertion point. Activity takes place at the insertion point even if that point is not visible. If, for example, you open a folder, scroll to the end of the file, then start typing, the characters you type are inserted at the start of the file (where the cursor is), rather than at the end of the file (which you are merely looking at).

To mark text to be deleted or copied, set the insertion point to the start of the text (move the mouse there and click), then drag the mouse through the text to be acted upon. Selected text is displayed in inverse video. Click twice to select a word, three times to select an entire line. To select large pieces of text, put the cursor at the start of the text, move the mouse to the end of the text, and shift click at that point.

At any time you can,

Open a new folder
(select the PULL item in the DESKTOP menu)

Start editing in any folder on the screen
(select the desired folder from the tray icon menu, or click in the body of the desired folder)

Move the folder around on the screen
(drag the folder's tab)

Make the folder larger or smaller
(drag the grow box. The grow box is the square in the lower right corner of the folder)

Scroll up a line
(click the up arrow box in the lower right corner. To scroll continuously, hold the button down in the box)

"LisaPascalDevManFeb82_059.PICT" 85 KB 2000-12-12 dpi: 72h x 72v pix: 1331h x 1998v

Scroll down a line
(click the down arrow box in the upper right corner. To scroll continuously, hold the button down in the box)

Jump back a windowful
(click in the grey area above the elevator. Hold the button down in this area to continue flipping pages. The elevator is the empty box in the vertical scroll bar)

Jump forward a windowful
(click in the grey area below the elevator. Hold the button down to continue flipping pages)

Jump to certain place in the folder
(drag the elevator to the position in the scroll bar that corresponds roughly to the desired position in the file)

Cut out the selected text and place it in the Scrap
(select the CUT item in the EDIT menu, or type Command-Z)

Paste the Scrap contents into the folder at the selection point
(select PASTE in the EDIT menu, or type Command-X)

Copy the selected text into the Scrap
(select COPY in the EDIT menu, or type Command-C)

Adjust the selected text right one space
(select ADJUST RIGHT in the EDIT Menu, or type Command-R)

Adjust the selected text left one space
(select ADJUST LEFT in the EDIT Menu, or type Command-L)

Save all your edits and close the folder
(select PUT BACK in the DESKTOP menu)

Save all your edits, but remain in the folder
(select ACCEPT ALL EDITS in the DESKTOP menu)

Write the current folder contents to another file
(select CROSSFILE TO... in the EDIT Menu. CrossFile asks you for the file name to cross file to. If that file already exists, you are given a chance to change your mind before the old file is overwritten. CrossFile does not change the file name used by Accept All Edits or Put Back. If you do not want to crossfile after all, type <RETURN> as the filename).

Cancel all the editing done since the last save command
(Select UNDO ALL EDITS from the DESKTOP menu. The editor gives you a chance to change your mind before it cancels all your edits).

"LisaPascalDevManFeb82_060.PICT" 79 KB 2000-12-12 dpi: 72h x 72v pix: 1308h x 1996v

Exit from the Editor
(Select EXIT EDITOR from the DESKTOP menu. If there are unsaved edits in the folder, the editor asks you if these should be thrown away. The prompts force you to answer "Y" then "N" or vice versa to be able to get out, which is less than friendly.)

Set tab stops
(select SET TABS... from the EDIT Menu. You can change the number of spaces between tab stops. The default is eight)

Find some target string starting from the current selection
(select FIND... from the SEARCH Menu. The default search ignores case and is token oriented. To change either of these, select the appropriate item in the SEARCH menu. FIND asks you for the target string. To find the same thing again, select FIND SAME. FIND & PASTE ALL performs a global find and replace. FIND can be invoked by Command-F, and FIND SAME can be invoked by Command-S. Only the first eight characters of a token-oriented search target are significant).

To move text from one folder to another, select and COPY the text from the source folder, activate the destination folder, set the cursor to the desired insertion point, and select PASTE.

CUSTOMIZING THE EDITOR

The editor uses whatever font it finds in the file LISA:EDITOR.FONT to display the folder contents. The suggested fonts are:

TITLE12R12S.F	20 lines x 82 chars
SARA8.F	26 lines x 83 chars
TILEX.F	32 lines x 82 chars (default)
TILE7R15S.F	32 lines x 94 chars
TILE5R18S.F	37 lines x 132 chars

"LisaPascalDevManFeb82_061.PICT" 67 KB 2000-12-12 dpi: 72h x 72v pix: 1324h x 1989v

THE UCSD EDITOR

File needed: UCSDEDITOR.OBJ

INTRODUCTION

The Editor is divided into the modes listed in the Editor's command line. A mode is invoked by typing the first letter of its name. Many of these modes also have command lines to further specify what action the Editor is to take. To insert new text, for example, you must type I when at the top level. Once in Insert mode, text can be inserted at that point in the file simply by typing it in. To accept the insertion type Ctrl-C (ENTER). To forget the insertion, type <ESCAPE> (the CLEAR key in the numeric keypad). To get to another mode (to move to a different place in the file, for example), you must first exit the mode you are in.

The entire file being edited must be in memory at all times during editing. There is no macro facility. The cursor is sometimes not where it appears to be. If a key is held down for more than a fraction of a second, the key's action is repeated at about 10 times per second.

Elaborate documentation can be found in the Apple /// Introduction, Filer, and Editor Manual or in the Apple][Pascal Operating System Reference Manual. The Apple][manual is somewhat inaccurate because our Editor is an enhanced version of the Pascal 1.1 update of the original Editor.

CURSOR MOVEMENT

Numerical Arguments

Most commands can be preceded by a numerical argument. The argument can be an integer between 0 and 9999, or the character / which means "as many times as possible". The numerical argument defaults to 1. It is abbreviated as "n" below.

Set Direction

The Set direction is the direction in which some commands move through the file. The first character in the command line tells what the current direction is: > is forward, < is backward.

At the top level of the Editor, the keys

> . +

change the direction to forward, and

< , -

change it to backward.

"LisaPascalDevManFeb82_063.PICT" 77 KB 2000-12-12 dpi: 72h x 72v pix: 1297h x 1983v

Simple Cursor movement

<SPACE>	move n spaces in set-direction
<RETURN>	move n lines in set-direction (you end up at the start of the line)
Up-Arrow	move n lines up
Down-Arrow	move n lines down
Left-Arrow	move n spaces left
Right-Arrow	move n spaces right
<TAB>	move n tab positions in set-direction
<=>	move to start of last text replaced, found, or inserted
0	move to start of line
$	move to end of line
W	move to beginning of next word
E	move to end of current word
B	move backward one word
T	move to the next occurrence of a specified character

You can also use the main keyboard "h", "j", "k", and "l" keys for cursor movement:

h	move left
j	move down
k	move up
l	move right

G(oto

Jump to some place in the file. You are presented with a secondary command line to tell the cursor where to Jump:

B(eginning of file
E(nd of File
M(arker

Jump to a specific marker. You can have up to 10 active markers set by the Set command (see below). Markers are pointers to file, not text, locations. You can return to the point you jumped from if you immediately type <ESCAPE>.

P(oint

like <=> above.

<ESCAPE> exits without Jumping.

P(age

Move the cursor n windows in the set-direction.

F(ind

Find a target string and place the cursor just after it. The normal mode is case sensitive. With the exceptions of S, L, U, and T, the first character following the F (for FIND) is the delimiting character. The target string specification ends when the delimiting character is repeated. Thus F.hi. looks for one

"LisaPascalDevManFeb82_064.PICT" 64 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1991v

occurrence of the lowercase word "hi". /F(HI(places the cursor after the last occurrence of the uppercase word "HI".

L(iteral, T(oken
: FIND can be told to look for any occurrence of the target string (Literal mode), or only for delimited occurrences (Token mode). The default is the one not given as an option when you type F (usually Token mode is the default). To override the default for one search, type L or T after F, but before the delimiting character. FL.hi. looks for any occurrence of the two characters "hi" in the file. A delimiter (for Token mode) is any character which is not a letter or a number. See S(et E(nvironment below to change the default mode. Note also that FL/<RETURN><RETURN>/ finds the next blank line.

U(pper-and-Lower-Case
: To get a case insensitive search, type U before the delimiting character. FU.hi. looks for any occurrence of "HI", "hi", "Hi", or "hI" in the file.

S(ame-String
: To repeat a search using the same target string as was used in the last search, you can type S in place of the delimited string. The Environment information (see S(et E(nvironment) tells you the current target string. If we last searched F.hi., FS searches again for "hi". Mode changing letters can be concatenated: FLUS looks for the target string last specified, in Literal mode without sensitivity to upper or lower case.

<ESCAPE>
: exit Find mode without taking any action.

TEXTUAL COMMANDS

I(nsert
: Insert text into a file. Insertion takes place between the character on which the cursor is sitting and the character immediately to its left. Ctrl-C accepts an insertion, <ESCAPE> forgets it. At any point during an insertion, left arrow deletes the character to the left of the cursor. You can't backspace past the beginning of the insertion. If you accidentally hit <ESCAPE> during an insertion (and thereby delete everything you just typed), you can rescue the inserted text by C(opying it from the B(uffer.

"LisaPascalDevManFeb82_065.PICT" 81 KB 2000-12-12 dpi: 72h x 72v pix: 1316h x 1997v

D(elete

Delete text moving in the Set-direction. Ctrl-C accepts a deletion, <ESCAPE> restores the deleted text. All deleted text is stored in the Copy Buffer, so to move a piece of text from one place to another, first delete it, move the cursor to the new location, then C(opy from the B(uffer. D(elete accepts numerical arguments (including / and P(age).

Z(ap

Delete all text between the current cursor position and the current location of the "Point" (the location of the last text inserted, found, or replaced). Zapped text is placed in the Copy Buffer.

C(opy

Copy text into the file at the current cursor position. The copied text can be taken from the Copy Buffer, or from a file.

F(ile

Copy a file, or some portion of that file. If the file has markers placed in it, you can choose to copy only the portion between any two markers by enclosing the names of the markers in square brackets after the file name. HI.TEXT[PROC1,PROC3] copies the portion of HI.TEXT lying between the markers PROC1 and PROC3. [,MARKER] copies from the beginning of the file to MARKER, and [MARKER,] copies from MARKER to the end of the file. If no marker specification is given, the entire file is copied.

B(uffer

Copy the contents of the Copy Buffer.

M(arker

Copy text between two markers, using exactly the same syntax as that used in copying from a file.

<ESCAPE>

Exit Copy mode without copying anything.

X(Change

Exchange mode allows you to overwrite text (exchange old text for new). You cannot exchange characters past a <RETURN>. Ctrl-C accepts an exchange, <ESCAPE> forgets it. Right arrow copies characters as it passes over them.

R(eplace

R(eplace mode is very similar in syntax to the F(ind mode. R(eplace takes two strings--the target string (as in F(ind), and the string to substitute for an occurrence of the target. Numerical arguments are valid.

L(iteral, T(oken

See F(ind above.

U(pper-and-Lower-Case

"LisaPascalDevManFeb82_066.PICT" 80 KB 2000-12-12 dpi: 72h x 72v pix: 1310h x 2004v

See F(ind above.

S(ame-String
See F(ind above. Note that S can be used for either or both strings: RSS looks for the previous target string and replaces it with the previous substitute string.

V(erify
If the find-and-replace operation is done in V(erify mode, you are asked before each replacement whether the replacement should take place for the given occurrence of the target. /RV.hi./ho/ looks in the set-direction for every occurrence of "hi", then on each one asks you if you really want to replace that "hi" with "ho".

<ESCAPE>
Exit Replace mode without replacing anything.

A(djust
Adjust the indentation of a line, or a group of lines. Left and Right Arrow push the entire line left or right. <SPACE> moves a line a space in the set direction. Up and Down Arrow cause the same adjustment to affect lines above or below the first adjusted line. <RETURN> affects the lines above or below, depending on the set direction. <TAB> moves lines over 8 spaces.

Alternate choices are:

L(eft-justify
R(ight-justify
C(enter
P(age

Ctrl-C accepts the adjustment--you have no other choice. Numerical arguments (including /) can be used.

M(argin
The M(argin command attempts to fit a paragraph into the prevailing paragraph formatting parameters set in the E(nvironment. M(argin only works when I(ndent-Auto is false and F(illing is true. A paragraph is defined to be any text bounded above and below by two delimiters. The allowed delimiters are a blank line, the beginning of the file, the end of the file, and a line which begins with the Command Character. To M(argin a paragraph, move the cursor to any point within the paragraph and type M.

O(bscure
O means "join" for some reason. It replaces the <cr> at the end of the current line with a space, thereby joining the two lines.

"LisaPascalDevManFeb82_067.PICT" 76 KB 2000-12-12 dpi: 72h x 72v pix: 1309h x 1987v

EDITOR DIRECTIVES

S(et

S(et mode allows you to set Markers and to change various global Editor settings.

M(arker

A Marker is an absolute location in a file to which you can J(ump. A maximum of ten markers can be active at one time in a file. To set a marker, move the cursor to the desired location, then type SM. You are then asked for the marker's name (a character string of up to 8 characters).

E(nvironment

The editing "environment" can be customized to suit your whims. The E(nvironment command displays the current settings of various parameters, the current size of the file, the current target and replacement strings for F(ind and R(eplace, the markers currently in the file, and various other information.

To change the setting of the parameters, type the first letter, then the new value. SEAF, for example, enters S(et mode, enters the E(nvironment menu, then sets A(uto-Indent to False. To exit the Environment menu and retain the changed values, type Ctrl-C (ENTER).

A(scii-file

Used by BASIC on Apple ///. Not applicable to Lisa.

C(ommand Character

You can set the Editor Command Character to be any printing character. The command character can be used to protect a line from the Margin command.

F(illing

If Filling is true, lines are automatically broken at the right margin between words.

I(ndent-Auto

If I(ndent-Auto is true, a new line is automatically indented to the position of the first non-space character of the previous line.

L(eft, R(ight, P(aragraph margin

These three parameters control the layout of the paragraphs when using Filling or Margin. P(aragraph margin is the indentation of the first line of the paragraph.

N(ame-of-file

When you Q(uit the Editor, the S(ave option can be used to write the edited file to the file named in

"LisaPascalDevManFeb82_068.PICT" 75 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1997v

the N(ame field. You can change this name to any legal file name.

S(pace-continue
If TRUE, a space must be typed before you can continue editing after an error has occurred. If FALSE, any character can be typed to allow you to continue.

T(oken
T(oken defines whether the F(ind and R(eplace commands use the T(oken or L(iteral mode as a default.

<ESCAPE>
Restore all the parameter values that were in force when Set mode was entered, and exit Set mode.

V(erify
V(erify causes the Editor to redisplay the screen.

Q(uit
Exit the Editor or load a new file.

U(pdate
Update the "workfile". It is not an accident that this is the only mention of the "workfile".

E(xit
Exit the Editor without saving anything that you did.

R(eturn
Return to the Editor (in case you didn't want to quit).

S(ave
Save the current edited file under the name given when the Editor was entered, or (if you started a new file) under the name last given in a W(rite command. Return to Q(uit menu.

W(rite
Write the current edited file to a diskette file. You must give the file's name. W(rite returns to Q(uit menu.

C(hange
Load a new file into the Editor without restarting the Editor. The previous target and substitution strings are carried over, and the copy buffer is maintained intact (if possible).

"LisaPascalDevManFeb82_069.PICT" 57 KB 2000-12-12 dpi: 72h x 72v pix: 1304h x 1994v

UTILITY PROGRAMS

(IUManager, ChangeSeg, SegMap) 72

(Configure, Contrast, SetSP, ChangeMem, Flip4, MoveSoroc). . . . 75

(FileDiv, FileJoin) 77

(Diff, FindID, Pretty List, PascalRef) 80

(DumpObj, DumpHex, Patch, ObjDiff, ByteDiff, GxRef). 87

(LisaTest) 91

(Perform, Coverage Analysis) 93

(Script) 96

(Terminal Emulator). 98

"LisaPascalDevManFeb82_071.PICT" 30 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1987v

IUMANAGER

IUMANAGER modifies the file INTRINSIC.LIB used by the intrinsic unit Linker and loader to find the intrinsic unit files. INTRINSIC.LIB is essentially a directory of unit names, segment names, and file names. When executed, IUMANAGER asks for the input and output files to be modified. The default name for both files is *INTRINSIC.LIB. The intrinsic unit manager has three modes: Manage Segments, Manage Units, and Manage Files. Each mode operates on an associated table of information used by the loader to properly link intrinsic unit code and data into an executing program.

Manage Segments Mode operates on the Segment Table which contains a list of segment names with information about each segment for the loader and linker. Manage Units operates on the Unit table which contains the unit names and information used by the loader and linker to build the data pointer table. Finally, Manage Files operates on the File Table which contains a list of files indexed by a file number. The loader uses the file number to find the intrinsic units and segments on the disk.

IUMANAGER has the following commands:

Q(uit	Write the output file and exit from IUMANAGER.
S(egments	Enter the Segment Manager and list the contents of the Segment Table.
U(nits	Enter the Unit Manager and list the contents of the Unit Table.
F(iles	Enter the File Manager and list the contents of the File Table.

In each of the modes you can:

L(ist	List the contents of the currently active table. If you have more than 32 entries in the table, you can stop the listing with Control-S (the '=' key on the numeric keyboard).
R(emove	Remove an entry from the currently active table.
C(hange	Modify information in the currently active table. The Change command prompts you for the value of each field. A response of <cr> accepts the default.
N(ew	Create an entry in the currently active table. The New command prompts you the value of each field. A response of <cr> accepts the default value.

"LisaPascalDevManFeb82_072.PICT" 82 KB 2000-12-12 dpi: 72h x 72v pix: 1310h x 1998v

In the File Manager you have one further command choice:

I(nstall — Install (update) the segment and unit tables from a linked object file. The Install command prompts you for the file number of the entry to be updated.

"LisaPascalDevManFeb82_073.PICT" 20 KB 2000-12-12 dpi: 72h x 72v pix: 1297h x 1963v

CHANGESEG

CHANGESEG changes the segment name in the modules in an object file. The first prompt asks for the object file you want to change:

```
File to change:
```

Changes are made in place (the file itself is changed). You are next asked:

```
Map all Names (Y/N)
```

If you want to change segment names in all modules, respond Y. If you want to be prompted for the new segment name for each module, type N. A response of <cr> accepts the default name.

SEGMAP

SEGMAP produces a segment map of one or more object files. The first prompt:

```
Files to Map ?
```

accepts either an object file name or a command file name. A command file must be preceded with a <. SEGMAP adds the .TEXT suffix to the command file name. The next prompt:

```
Listing File ?
```

directs the map information to the file given. A response of #1: or CONSOLE:, for example, send the map information to the screen. The map information includes the object file name, the name of the unit in the file, the names of the segments used in that unit (if any), and the new segment names.

"LisaPascalDevManFeb82_074.PICT" 48 KB 2000-12-12 dpi: 72h x 72v pix: 1308h x 2000v

CONFIGURE

CONFIGURE modifies some of the vectors in the Monitor Map Table. These vectors are stored in CONFIG.DATA on the male boot volume and are used by the Monitor to configure your system when it is booted. To use CONFIGURE, copy CONFIG.DATA from the male side to a female volume, or flip the sex of the boot diskette. X(ecute CONFIGURE. CONFIGURE asks you whether it should Go or Quit. Type G to run CONFIGURE, Q to return to the Monitor command line. CONFIGURE asks you for the file containing the vectors you want to change. If you do not give a volume name, it look on the prefix volume. If you give just the volume name, it looks for CONFIG.DATA on that volume. CONFIGURE can change the following vectors:

```
dE(bug pointer           [$150]
D(efault Stack pointer   [$13C]    (* most important *)
H(eap pointer            [$138]
C(orvus pointer          [$134]
U(art pointer            [$11C]
A(pple port              [$118]
M(emory top              [$114]
S(creen base             [$110]
B(uffer pointer          [$10C]
```

The old and new value of each vectors is also displayed. Type the capitalized letter of the vector you want to change. Lower case is allowed in hexadecimal numbers. When you are done, type Q to Quit. At this point you are asked where to save the new values. You can write the changes back to CONFIG.DATA, exit without making any changes, and so on.

A memory map is given in the Monitor chapter showing the relationship of these vectors. Do not place the start of the heap above the stack pointer. Because CHANGEMEM sets aside heap space, it is safer to set the stack pointer before grabbing a lot of the heap with CHANGEMEM.

Once you have finished modifying CONFIG.DATA, transfer it back to the male boot volume so that it can take effect when the system is rebooted.

CONTRAST

CONTRAST changes the contrast setting of your screen without changing the default setting. It is a simple program that should be self-explanatory. Like CONFIGURE, it first asks whether to G(o or Q(uit. If you type G, some alphanumeric characters are scattered around the screen for reference. You can type '>' or '.' to increase and '<' or ',' to decrease the screen contrast.

"LisaPascalDevManFeb82_075.PICT" 87 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1981v

SETSP

SETSP sets the address at which the stack pointer starts. Memory above that address is then reserved for code, and memory below it is the stack and heap space. If your program requires a great deal of room for data, set the stack pointer to a high address. If the program requires a great deal of code, set the SP to a low address. The Monitor default SP starting address depends on the version of CONFIG.DATA on your male boot volume. The highest possible address is the bottom of the Monitor.

All SETSP I/O is in hex. When X(ecuted, it displays the current stack pointer value. Type 0 to exit the program. The optimal value to give SETSP may not be obvious at first, since code swapping can change your memory requirements. It is quite possible that a program will run happily for hours, then die with a Loader Error when a piece of code couldn't be fit in memory. If this happens, set the stack pointer to a lower starting address, and try again.

CHANGEMEM

CHANGEMEM changes the size of the predeclared RAM-resident volume MEMORY:. Its interface is identical to that of SETSP. The default size of MEMORY: is 10 blocks. Space for the MEMORY: volume is taken from the available heap space.

FLIP4

Volume #4: is normally the RAM-based volume MEMORY:. The Disk drive that would usually be #4: is hidden from the monitor to avoid overwriting the male boot volume. If you want access to that disk drive from the monitor, run FLIP4. FLIP4 executes a simple loop until you tell it to Quit. After asking whether to continue or to quit, FLIP4 gives you a chance to toggle the state of #4:. #4: is either MEMORY: or the disk drive. Remember to remove the male boot volume before writing to #4:.

MOVESOROC

MOVESOROC (also sometimes known as MS) determines where the Pascal WRITELN output goes. It normally is sent to the Lisa screen. When an application is running on this screen, however, debugging WRITELNs mess up the program's pretty output. MOVESOROC redirects this output to either the Apple monitor, the UART (serial port #2), or back to the Lisa. Monitor input always comes from the terminal to which output has been directed.

"LisaPascalDevManFeb82_076.PICT" 88 KB 2000-12-12 dpi: 72h x 72v pix: 1305h x 1993v

FILEDIV and FILEJOIN

It is often necessary to distribute files that are too large to fit onto a single floppy diskette. FILEDIV can be used to break a large file into several diskette-sized pieces. FILEDIV can then be used to rejoin these pieces at the file's destination. These two programs replace the TRANSFER program.

To divide a large text or object file, execute FILEDIV.

```
Input file:  <give the name of the file to be divided>
Output file: <give the name to be used for the output files>
```

Do not include the suffix in the file name. If, for example, you want to divide TEMP.TEXT, give TEMP as the input file, and TEMP (or whatever) as the output file. FILEDIV will create a group of files named TEMP.1.TEXT, TEMP.2.TEXT, and so on, until TEMP.TEXT is completely divided up. If you use the drive number (#9:, for example), rather than the volume name, the new files can be written to multiple diskettes. When space on a diskette is exhausted, FILEDIV asks you to insert another diskette.

To rejoin the pieces of the file, execute FILEJOIN. Using the example given above, we can rejoin TEMP.1.TEXT and friends into TEMP.TEXT by responding:

```
Input file: TEMP              <will read TEMP.1.TEXT, etc>
Output file: TEMP             <will create TEMP.TEXT>
```

FILEDIV and FILEJOIN use regular directories, so a spurious sex change cannot destroy your file. Files are verified in both directions.

"LisaPascalDevManFeb82_077.PICT" 61 KB 2000-12-12 dpi: 72h x 72v pix: 1307h x 1985v

DIFF

DIFF is a program for comparing ".TEXT" files, in the LISA Pascal development environment. DIFF is strongly oriented toward use with Pascal or Assembler source files.

DIFF is not sensitive to upper/lower case differences. All input is shifted to a uniform case before comparison is done. This is in conformance with the language processors, which ignore case differences.

DIFF is not sensitive to blanks. All blanks are skipped during comparison. This is a potential source of undetected changes, since some blanks are significant (in string constants, for instance). However, DIFF is insensitive to "trivial" changes, such as indentation adjustments, or insertion and deletion of spaces around operators.

DIFF does not accept a matching context which is "too small". The current threshold for accepting a match is 3 consecutive matches. The M option allows you to change this number. This has two effects:

> Areas of the source where almost "every other line" has been changed will be reported as a single change block, rather than being broken into several small change blocks.
>
> Areas of the source which are "entirely different" are not broken into different change blocks because of trivial similarities (such as blank lines, lines with only "begin" or "end", etc.)

DIFF makes a second pass through the input files, to report the changes detected, and to verify that matching hash codes actually represent matching lines. Any spurious match found during verification is reported as a "JACKPOT". The probability of a JACKPOT is very low, since two different lines must hash to the same code at a location in each file which extends the longest common subsequence, and in a matching context which is large enough to exceed the threshold for acceptance.

DIFF can handle files with up to 2000 lines.

DIFF first prompts you for two input file names: the "new" file, and the "old" file. DIFF appends ".TEXT" to these file names, if it is not present. DIFF then prompts you for a filename for the listing file. Type carriage-return to send the listing to the console.

DIFF does not (currently) know about INCLUDE files. However, DIFF does allow the processing of several pairs of files to be sent to the same listing file. Thus, when DIFF is finished with one pair of files, it prompts you for another pair of input files. To terminate DIFF, simply type carriage-return in response to the prompt for an input file name.

The output produced by DIFF consists of blocks of "changed" lines. Each block of changes is surrounded by a few lines of "context" to aid

"LisaPascalDevManFeb82_078.PICT" 104 KB 2000-12-12 dpi: 72h x 72v pix: 1315h x 2000v

in finding the lines in a hard-copy listing of the files.

There are three kinds of change blocks:

INSERTION -- a block of lines in the "new" file which does not appear in the "old" file.

DELETION -- a block of lines in the "old" file which does not appear in the "new" file.

REPLACEMENT -- a block of lines in the "new" file which replaces a corresponding block of different lines in the old file.

Large blocks of changes are printed in summary fashion: a few lines at the beginning of the changes and a few lines at the end of the changes, with an indication of how many lines were skipped.

DIFF has three options which allow you to change the number of context lines displayed (+C), the number of lines required to constitute a match (+M), and the number of lines displayed at the beginning of a long block of differences (+D). To set one of these numbers, type the option name followed by the new number to the prompt for the first input file name. +D 100, for example, causes DIFF to print out up to 100 lines of a block of differences before using an ellipsis. The maximum number of context lines you can get is 8.

"LisaPascalDevManFeb82_079.PICT" 52 KB 2000-12-12 dpi: 72h x 72v pix: 1299h x 1972v

FINDID

FINDID searches code files for an identifier. It provides a service similar to that of the editor's literal search, but the search can cover any number of files of any size. When executed, it asks first for the name of the file which contains the list of files through which you want to search. For example, if you want to search the files CODE.TEXT, CODE1.TEXT, and CODE2.TEXT, make a file which contains:

```
Code.Text
Code1.Text
Code2.Text
```

and give FINDID this file's name. FINDID then asks for the identifier you want to search for. Only the first eight characters are significant. The search is always literal--any identifier beginning with the specified eight characters is considered a match. FINDID's last prompt asks whether the search should ignore case differences. FINDID then grovels through files in the list reporting any occurrences of the identifier. To get out of FINDID, hit NMI, then type RM to LisaBug.

"LisaPascalDevManFeb82_080.PICT" 45 KB 2000-12-12 dpi: 72h x 72v pix: 1312h x 2000v

PRETTY LIST

Pretty List scans a listing produced by the Assembler, and replaces the asterisks in the displacement portion of branch instructions with the actual forward reference value. When you X(ecute PRETTY, you are asked for the Input File (the Assembler listing file). Because this file can be either a data file or a text file (with a .TEXT extension), the next prompt is:

```
        If input file is a text file (file.text) type 1 else type 0 --
```

Pretty List then asks for the output file name.

If the listing file contains:

```
0360|                   CHKLO   BSR4    CHKMEM
0360| 49FA ****   #            LEA     @1,A4
0364| 6000 ****   #            BRA     CHKMEM
0362* 0006
```

Pretty list produces:

```
0360|                   CHKLO   BSR4    CHKMEM
0360| 49FA 0006   #            LEA     @1,A4
0364| 6000 005A   #            BRA     CHKMEM
```


"LisaPascalDevManFeb82_081.PICT" 39 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1984v

PASCALREF

Pascalref is a cross reference utility for Lisa Pascal programs. It can perform partial or complete cross references, can handle USES and INCLUDE statements correctly, and imposes no limit on the size of the target program.

Pascalref assumes that the program or unit to be referenced (target program) has been compiled without syntax errors. It also assumes that the font BOLD10V and the file MPMENUFILE.TEXT are available on your prefix volume or the boot volume (#5:).

THE USER INTERFACE

ACTION	SEARCHOPTIONS	FINDTYPES	YN-TF	<-- The Menu Bar
Setup Files	Interactive	Declared	Yes-True	
Set Scope	Reloffsets	Modified	No-False	
Begin Pascalref	Procdic	Accessed		
	Widepaper	Stnd PFT		
	Used Unit Int			
	Out Scope Vars			

The line in capitals at the top represents the menus in the menu bar. The lower case names are the items in each pull down menu. A shaded menu item shows that the option represented by that item is active. To change the 'Procdic' option, for example, use the mouse to select 'Procdic' then select either True or False. You can also type SP, then enter Y(es or T(rue, N(o or F(alse. When you have all your options set up, select 'Begin Pascalref'.

OPTIONS

Setup Files

You are asked for the names of the listing file, source file, and output file.

Set Scope

Pascalref allows you to set the scope to be a single procedure. Only identifiers within that procedure and its local procedures are referenced. Of course, accesses to any variables global to the procedure are included in the OUT OF SCOPE section. The default scope is the whole program. Currently, when referencing a Unit by itself, only set the scope to the whole unit. Also, don't set the scope to a FORWARD procedure or a procedure in the interface of a unit.

"LisaPascalDevManFeb82_082.PICT" 76 KB 2000-12-12 dpi: 72h x 72v pix: 1367h x 2000v

Begin Pascalref

Start the cross reference.

SEARCHOPTIONS

Interactive

When Interactive is true (the default), Pascalref looks only for those names that occur in the list of variables that you type in.

When a particular reference is finished, Pascalref asks you 'Look at more identifiers?'. If you answer yes, PascalRef returns to the options setting routine with the same files set up. You can change the options and the files on each pass if you want to.

When Interactive is false, Pascalref cross references all identifiers within the specified scope.

Reloffsets

Next to each occurrence of an identifier is the line number it occurs on. When Reloffsets is true (the default), the line numbers are listed relative to the procedure the occurrence is in.

When Reloffsets is false, offsets are given relative to the beginning of the program.

Procdic

If Procdic is true, Pascalref creates a procedure dictionary listing each Procedure with the line it starts on. The default for ProcDic is false. The program or outermost procedure in the scope is procedure #1. Nested procedures are indented.

In the case of forward procedures and procedures declared in the interface of a unit, the procedure number listed reflects where the procedure declaration occurs. '=Forward Proc' appears in the procedure dictionary after the 'STARTING LINE #'. The ordered location of the procedure name and the starting line number within the procedure dictionary reflect where the header to the body part of the procedure occurs.

Widepaper

The default value (False) should be used when sending output to the console or to standard 8-1/2 inch wide paper. When printing on 132 column paper, set Widepaper to true.

"LisaPascalDevManFeb82_083.PICT" 71 KB 2000-12-12 dpi: 72h x 72v pix: 1311h x 1982v

Used Unit Int

The default value of True causes the INTERFACE parts of USED units to be included in the PASCALREF scan. This allows you to see where every identifier used by a Pascal program is defined. If you are not interested in the INTERFACE of a unit, set 'Used Unit Int' to false.

Out Scope Vars

When a program accesses a identifier that has not been defined, that access shows up in the OUT OF SCOPE VARIABLES OR PROCEDURES section of the PASCALREF listing. This part of the listing is present when 'Out Scope Vars' is true (the default).

FINDTYPES

The three 'Find Types' are DECLARED, MODIFIED and ACCESSED. The default value of each type is true. Each type can be set independently. When all are true, all occurrences of an identifier are listed. If, for example, only ACCESSED is true, PascalRef lists only places where a variable is accessed. MODIFIED flags places where a variable occurs to the left of :=, and where it is passed as the actual parameter to a formal VAR parameter. DECLARED lists places where an identifier is declared.

Stnd PFT lists occurrences of standard procedures, functions and types. Its default is false.

YN-TF

This menu gives you an option for answering Yes-No and True-False questions. Choosing Yes-True is the same as entering a 'Y' or 'T' from the keyboard and choosing No-False is the same as entering a 'N' or 'F'.

"LisaPascalDevManFeb82_084.PICT" 61 KB 2000-12-12 dpi: 72h x 72v pix: 1313h x 1993v

OUTPUT FORMAT

LISTING (Pascalref can produce a listing identical to that produced by the compiler)

PROCEDURE DICTIONARY:

```
  PROC #,  PROC NAME,                                  STARTING LINE #
  -----    ---- ----                                   -------- ----

    1:     MAINPROGRAM                                    0
    3:       NESTEDPROC                                  80
    4:         FURTHERNESTEDPROC                        120
    5:       PROC                                       200
    2:       FORWARDPROCWHOSEBODYCOMESAFTERPROC          40=Forward Loc
    6:         PROCNESTEDINBODYOFFORWARDPROC            280
    7:       LASTGLOBALPROC                             300
```

```
The identifier we are referencing
|            The procedures it occurs in
|            |           The occurrences within those procedures
|            |           |
IDENTIFIER   PROCEDURE   OCCURRENCE  (D=defined, A=accessed, M=modified,
----------   ---------   ----------   V=Var param def, P=passed to Var param)

I            PROCA       D    10, P    45, A    50, A    60.
             PROCB       D    16, A    20, A    30,

STRNG        PROCA       D    12, M    41, M    62, A    50, A    58,
                         A    60,
             PROCC       V    75, M    80, A    82.
```

OUT OF SCOPE VARIABLES OR PROCEDURES

These are listed in the same format as that of the regular identifiers, but represent items that are global to the chosen scope. To find out what global variables a procedure and its nested procedures access, set the scope to the procedure, set INTERACTIVE to false and look at the resulting OUT OF SCOPE items.

GENERAL NOTES ON THE USE OF PASCALREF

Pascalref does not store information about variable types or record structures. If you have both a stand alone variable named AVAR and a record field named AVAR, PascalRef lists both as the same identifier.

Include commands are recognized by Pascalref and the INTERFACE parts of units USED by the target program are included in the

"LisaPascalDevManFeb82_085.PICT" 65 KB 2000-12-12 dpi: 72h x 72v pix: 1376h x 1996v

reference if the scope is set to the whole program.

To find variables that are declared but no longer used by a program, do a reference of the whole program. Variables that have a 'D' occurrence and no others can often be removed from the program.

Occurrences of a particular identifier are always exactly in order when interactive is true. When interactive is false, occurrences are grouped by the procedures where the identifier is declared locally. In the case where interactive is false, you may notice the following:

```
I           PROCA       D    10, P    45, A    50, A    60.
            PROCB       D    16, A    20, A    30,
```

The variable 'I' was declared and accessed in PROCA and declared and accessed in PROCB. The accesses in PROCA occur after the declaration and accesses in proc B but they are listed first.

If 'I' were not defined in PROCB, it would look like:

```
I           PROCA       D    10,
            PROCB       A    20, A    30,
            PROCA       P    45, A    50, A    60.
```

If the first example were done in interactive mode, it would look like:

```
I           PROCA       D    10,
            PROCB       D    16, A    20, A    30,
            PROCA       P    45, A    50, A    60.
```

Procedures and Functions as parameters are currently not fully implemented in Pascalref. They are parsed by Pascalref, but Variables passed to a procedure or function that is a parameter are always marked as modified in that occurrence.

"LisaPascalDevManFeb82_086.PICT" 57 KB 2000-12-12 dpi: 72h x 72v pix: 1313h x 1978v

Source: David T. Craig | Page 0086 of 0107

DUMPOBJ

DUMPOBJ is a disassembler for 68000 code. It can disassemble either an entire file, or specific modules (procedures) within the file. DUMPOBJ replaces DUMPMCODE.

DUMPOBJ first asks for the input file which should be an unlinked object file. The output (listing) file defaults to CONSOLE:. You are asked whether you want to dump

A(ll, S(ome, or P(articular modules.

If you respond S(ome, DUMPOBJ asks you for confirmation before dumping each module. A response of <ESC> gets you back to the top level. If you respond P(articular, DUMPOBJ asks you for the particular module(s) you want dumped.

The next question is: 'Dump file positions [N]?' The file position is a number of the form [0,000] where the first digit is the block number (decimal) within the file and the second number is the byte number (hexadecimal) within the block at which the module starts. This information can be used in conjunction with the PATCH program. Finally, DUMPOBJ asks if you want the object code disassembled.

"LisaPascalDevManFeb82_087.PICT" 49 KB 2000-12-12 dpi: 72h x 72v pix: 1302h x 1985v

DUMPHEX

DumpHex provides a textual representation of the contents of any file. The file dump is block-oriented with the hexadecimal representation on the left and the corresponding ASCII representation on the right. If a byte cannot be converted to a printable character, a dot is substituted.

When DumpHex is X(ecuted, it asks you for the name of the output file. A .TEXT extension is added if necessary. To direct the output to the console, type carriage return. After getting a valid output file name, DumpHex asks for the input file to be dumped. No extensions are appended, so give the full filename. Once a file has been completely dumped, DumpHex asks you for the next file to dump. Type carriage return to exit the program.

After opening the input file, DumpHex asks you which block to dump. The default (carriage return) is block 0. If the output is going to a file, you are asked which block is the last you want dumped. The default here (carriage return) is the last block in the file.

The format of the console output depends on the number of lines your screen has. If fewer than 33 lines are available, the output is displayed only a half block at a time. Between blocks or block halves you have the option to

Type <space> to continue, <escape> to exit.

Escape returns to the prompt for an input file.

"LisaPascalDevManFeb82_088.PICT" 60 KB 2000-12-12 dpi: 72h x 72v pix: 1315h x 1993v

PATCH

Patch allows you to examine and change the contents of any file. The display of the file's contents is exactly like that of DumpHex. With Patch, however, you can use the cursor control keys to move around in the block and change the value of any byte using either the hexadecimal representation on the left or the ASCII representation on the right.

After X(ecuting Patch you are asked for the full name of the file to patch. Carriage return exits Patch. No extension is appended to the file name. You are then asked for the number of the block you want to mess around with. Carriage return here returns you to the file name prompt.

The block is displayed with the cursor in the upper left corner at word 0 of the block. The arrow keys can be used to move around in the block. If you move the cursor up from the top line, you get the bottom line of the preceding block. Similarly, if you move down from the bottom line, you move into the top line of the next block.

When the cursor is on the hexadecimal side of the display, you can change any byte by typing the new hexadecimal value. Any non-hex characters are ignored. You can impress your friends by pointing out that the change is reflected automatically in the ASCII portion of the display. When the cursor is on the ASCII side, type any character to replace the value of the byte. Until you move out of the block you can undo any changes by typing <escape>.

"LisaPascalDevManFeb82_089.PICT" 63 KB 2000-12-12 dpi: 72h x 72v pix: 1302h x 1987v

OBJDIFF

OBJDIFF performs a comparison of two object (.OBJ) files. The two files being compared should be very similar. OBJDIFF uses procedure boundaries to get itself back in sync after a difference is found.

BYTEDIFF

BYTEDIFF compares any binary files, but once it finds a difference between the two files, it does not always find where the differences end.

GXREF

GXREF lists all the modules which call a given procedure, and all the modules which that procedure calls. It provides a global cross reference of subroutines and modules.

"LisaPascalDevManFeb82_090.PICT" 32 KB 2000-12-12 dpi: 72h x 72v pix: 1319h x 1990v

LISATEST

LISATEST is a package of hardware test routines. The DIAG: diskette which contains these programs can be obtained from Rich Castro. To use the programs, boot the Apple II with the Lisa in the power-on reset state, then X(ecute LISATEST. You have eight choices:

1) Apple-Lisa Interface Test
2) Memory Test (RAMTEST)
3) Display Memory Test Results
4) UART Wrap Around Test
5) Video Latch Test
6) MMU Test
7) Keyboard Test
9) Quit

The tests are, for the most part, self-explanatory. For a complete description of each test, its prompts, error messages, and options please see the Lisa Production Tests documentation by Rich Castro. A short description of each test is given below.

Apple-Lisa Interface Test

The Interface Test attempts to use the parallel port interface between the Apple II and the Lisa to verify that the two systems can communicate with each other.

Memory Test

The Memory Test program tries to single out bad or marginal memory chips and provides trouble shooting information about other memory board problems. It is essentially an updated and easy to use version of RAMTEST.

Display Memory Test Results

After running the Memory Test, you can have the results of that test redisplayed by the Display Memory Test Results program.

UART Wrap Around Test

The UART Test checks the UART on the CPU board that controls the RS-232 port #1 (the one on the left as you face front of the machine). To run the test you need a specially wired wrap around DB-25 male connector.

Video Latch Test

The Video Latch Test checks the operation of the video page latch on the MCU board.

"LisaPascalDevManFeb82_091.PICT" 71 KB 2000-12-12 dpi: 72h x 72v pix: 1305h x 1985v

MMU Test

The MMU Test tries to verify that the MMU is working properly. It sets up the base and limit registers in the MMU with various values and then attempts to access the corresponding memory segments.

Keyboard Test

The Keyboard Test checks the keyboard and mouse buttons to verify that the COPS interfaces are functioning properly.

"LisaPascalDevManFeb82_092.PICT" 25 KB 2000-12-12 dpi: 72h x 72v pix: 1304h x 1985v

PERFORM

Perform monitors the execution performance of a program. After X(ecuting Perform, you are asked for the listing file's name. A carriage return directs output to the console. If necessary, .TEXT is appended to the listing file name. You are next asked for the name of the program you want to analyze. If necessary, the extension .OBJ is added to the file name. The program file must be executable and must be linked with the corresponding .DBG files.

Perform scans the program file for procedure entry points, listing them as they are found. It then waits for you to type a space before executing the program. Every 1/60 second the program's program counter is checked to find out which routine is executing at that moment. When the program terminates, Perform produces a listing of the routines it found executing ordered according to the amount of time spent in each routine. Routines that were never caught executing are listed separately. Perform may miss the execution of short or rarely called routines.

The longer the program runs, the more trustworthy the analysis. Routines that are synchronous with the 60 Hz clock are not measured correctly. If M.SYMBOLS is included, PERFORM also gives information on the amount of time spent in the monitor (MPASLIB).

"LisaPascalDevManFeb82_093.PICT" 58 KB 2000-12-12 dpi: 72h x 72v pix: 1301h x 1992v

COVERAGE ANALYSIS

The CA program provides a coverage analysis of a Lisa Pascal program. Branch counters are inserted into the source (.TEXT file) of the program under test. The output of the CA program is then compiled and linked. At the end of the program's execution, a text file is produced giving the branch numbers and the number of times each branch was executed.

To run the coverage analysis program you need to get the sources of any of the units and programs you want to analyze and the .OBJ versions of any other units that are required to link the program. You should be able to compile and link these sources without error. To add counters to a unit or program:

X(ecute CA

CA first asks for the name of the source file:

Input file -

Give it the name of the file you want analyzed. The output of CA is another source file containing the original source modified by the addition of the counters and the analysis machinery. The output file can be very large, so give it plenty of room.

Output file -

The name you give here is the name of the new source (with the branch counters added in).

The next prompt is:

Count B(ranches, P(rocedures, O(ne unit

You can count every branch (every THEN, ELSE, CASE branch, REPEAT, DO, etc), just procedure entries, or just report on the branch counters in a unit that has already been run through CA. If you ask that every branch be counted, CA also asks:

Routine to skip (<cr> for none):

The compiler has a fixed code buffer size. If a procedure in the original program is close to the size limit, it may be impossible to add counters to every branch and still compile that procedure. The compiler error is #350, "procedure too large". If you add the counters and the compiler complains about some procedure, run CA again, and give the offending procedure name here. If more than one procedure is too large, either complain to the developers, or ask someone to make the 'routine-to-skip' code take a list of names.

The next prompt was Pete Cressman's idea:

Enable tracing?

"LisaPascalDevManFeb82_094.PICT" 83 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1994v

Type 'y' and every time the program is executed you will be asked if you want to be informed about every procedure entry during execution of the program. This avalanche of names can be very tedious to watch. If you respond "n", the tracing machinery is omitted from the program and you are never asked whether it should be activated.

The next prompt is:

```
Data file -
```

The data file is the file containing the coverage analysis after the program has executed. It is a text file; so you can read it with the mouse editor. You can match each counter number up with the code it is counting by examining the output file that CA produces. At each branch you will find a procedure call of the form:

```
_CA_Inc(n);
```

where n is the counter number. All objects added by CA to the program start with the the four letters '_CA_' to try to avoid naming conflicts.

Finally, you are given a chance to place some arbitrary sequence of text in the header of the data file (date of the test, or whatever). Type <cr> to end the comments.

If you are adding counters to a unit, some of these questions are omitted because they are irrelevant.

Once CA has added the counters, you must compile the output file, generate the .OBJ file, and link it with all the units it requires. If all goes well, each time you execute the program the data file is updated. If the data file does not exist, it is created. If it does exist, the counter data it contains are added to the current counter values. The resulting data file therefore contains a record of an arbitrary number of program executions.

If you get the Linker warning:

```
Segment <mumble> too large
```

you will have to break that segment into two pieces, write a procedure to force the new segment to be resident whenever the old one was, and start over with the CA program. The problem here is that a segment can contain no more than 32 Kbytes of code or data. There is no way the CA program can tell when a segment is close to the limit. If a segment is right on the borderline, it is not inconceivable that the branch counters will cause it to overflow.

The counters pin at 32767. Programs that run in the Window Manager-OS environment are recognized, and theoretically correct code is issued, but no promises are made yet. CA increases the size of both the source .TEXT file and the final .OBJ file by about 30 percent. It may slow execution rates noticeably.

"LisaPascalDevManFeb82_095.PICT" 96 KB 2000-12-12 dpi: 72h x 72v pix: 1309h x 1990v

SCRIPT

SCRIPT is Colin McMaster's text formatting program. SCRIPT commands are:

Name	Default	Example	Effect
Page Length	66	^pl N	Define page length to be N lines
Page Number	1	^pn N	Start page numbering at N
Page Break	--	^bp	Start a new page
Need Lines	--	^ne N	Make sure at least N lines remain on page
Line Space	1	^ls 1	Set single or double spacing
Space	1	^sp N	Space N lines
Break Line	--	^br	Start a new line
Page Offset	0	^po N	Start leftmost printing at column N
Indent	0	^in N	Indent N columns from page offset
Temporary Indent	0	^ti N	Indent N columns for next line only
Right Margin	72	^rm N	Set line length to N characters
Fill	--	^fi	Set filling mode to true
No Fill	--	^nf	Set filling mode to false
Justify	--	^ad	Justify text to right margin
No Justify	--	^na	Turn justification off
Center	1	^ce N	Center next N lines
Text	--	^tx 'N	Display N literally
Change command	--	^cc N	Change SCRIPT command character to N
Title	--	^tl 'L'C'R	Print titles Left, Center, Right
Margin 1	4	^m1 N	Set number of lines above and including header
Margin 2	2	^m2 N	Set number of lines below header
Margin 3	2	^m3 N	Set number of lines above and including footer
Margin 4	4	^m4 N	Set number of lines below footer
Header	--	^he 'L'C'R	Place headers Left, Center, Right
Even Header	--	^eh 'L'C'R	Place even headers Left, Center, Right
Odd Header	--	^oh 'L'C'R	Place odd headers Left, Center, Right
Footer	--	^fo 'L'C'R	Place footers Left, Center, Right
Even Footer	--	^ef 'L'C'R	etc
Odd Footer	--	^of 'L'C'R	
Source	--	^so 'File'	Begin printing text of File
Zero Slash	--	^zs	Turn on zero slashing
No Zero Slash	--	^nz	Turn off zero slashing
Keywords	--	^kw	Underline Pascal keywords
No Keywords	--	^nk	Do not underline Pascal keywords
Define Macro	--	^de V0	Begin definition of macro V0
Terminate Macro	--	^^	End definition of macro
Append Macro	--	^am V0	Append to macro V0
Delete Macro	--	^dm V0	Delete macro V0

"LisaPascalDevManFeb82_096.PICT" 83 KB 2000-12-12 dpi: 72h x 72v pix: 1396h x 2003v

SCRIPT OPTIONS (specified when SCRIPT is executed)

```
-cC     Change command character to C
-fFILE  Send output to FILE (.TEXT is not appended for you)
-k      Underline Pascal keywords
-l      Assume output is going to a Printronix-style printer
-nN     Start page numbering at N
-oLIST  Output only the pages given in LIST
-p      Assume printer has full control of page
-q      Assume output is going to a Qume-like printer
-s      Stop printing after each page and wait for <cr> or <ESC>
-zN     Set page offset to N
```

This version of Script does not attempt to return to its top level when it has finished with a file. Because it is trying to exit the program from a unit, it usually quits with 'fatal error 1'. Do not use the options that refer to printers. To see your formatted text, use either -S or -F.

More complete documentation is available from Publications.

"LisaPascalDevManFeb82_097.PICT" 42 KB 2000-12-12 dpi: 72h x 72v pix: 1295h x 1963v

TERMINAL EMULATOR

Files needed: TERM.OBJ
LISA:SYSTEM.FONT
LISA:TERM.MENUS.
LISA:SARA8F

The terminal emulator (TERM) provides a Lisa folder which is a full duplex virtual terminal. The terminal control commands implemented here are similar to those of the Hewlett-Packard 2640 and 2645, the DataMedia, Perkin-Elmer Fox and Owl, Beehive, and the DEC VT-52 terminals, as well as the "VT52" modes of the DEC VT-100 and HeathKit H19 terminals.

The terminal emulator works only with the "new" Lisa hardware. In addition, you must make a hardware modification to your Lisa: open the back of the machine and find the three large chips in the center of the visible board (the IO board). The chip nearest the power supply should already have the 10th pin from the bottom on the power supply side raised. For the terminal emulator, the 9th pin from the bottom on the same side should also be raised.

To invoke the emulator, copy the files given above, and X(ecute TERM. The emulator has three menus and a tray icon. The Speed menu sets the baud rate. Available speeds are 300, 600, 1200, 2400, 4800, 9600, and 19200 baud. 600 baud is not available on Port #1. The default speed is 300 baud.

The Port Menu determines which serial port is connected to the modem. Port #1 is the connector in the center. Port #2 is the connector nearest the power supply. The default port is #2.

The control menu has four items: Record, Play Back, Debug, and Quit. If you select Record, all characters received by the UART are saved in the file RECORD.TEXT. If you select Play Back, the contents of the file PLAYBACK.TEXT are sent to the UART just as if they had been typed. If you want to see exactly what characters are being received, including control characters and escape sequences, select Debug. To exit the terminal emulator, select Quit.

The control commands are:

Ctrl-G	Bell (screen flashes)
Ctrl-H	Backspace
Ctrl-I	Tab (8 spaces)
Ctrl-J	Linefeed
Ctrl-M	Carriage return
Ctrl-[	Start Escape Sequence (see below)
Escape-@	Enter Insert Character Mode
Escape-A	Cursor Up
Escape-B	Cursor Down
Escape-C	Cursor Right
Escape-D	Cursor Left
Escape-E	Clear screen
Escape-H	Cursor home (top left corner)

"LisaPascalDevManFeb82_098.PICT" 91 KB 2000-12-12 dpi: 72h x 72v pix: 1308h x 1996v

Escape-I	Scroll down
Escape-J	Clear to end of screen
Escape-K	Clear to end of line
Escape-L	Insert line position
Escape-M	Delete line position
Escape-N	Delete character position
Escape-O	Leave Insert Character Mode
Escape-P	Insert character position
Escape-Y	Absolute character positioning (Y+31, X+31)
Escape-b	Clear to beginning of screen
Escape-j	Save Cursor position
Escape-k	Restore saved cursor position
Escape-l	Erase line
Escape-o	Clear to beginning of line
Escape-p	Stand out (bold characters)
Escape-q	Reset Stand Out (normal characters)
Escape-z	Initialize terminal

All other Escape and Control sequences are ignored.

"LisaPascalDevManFeb82_099.PICT" 35 KB 2000-12-12 dpi: 72h x 72v pix: 1310h x 1956v

ERROR MESSAGES

There are several error categories--I/O errors, Loader errors, trap handler errors, and Pascal Compiler errors. In most cases, you can type SPACE to return to the Monitor command line. Since nothing in the Monitor is tied to the user stack pointer, the Monitor can usually recover from errors that are fatal in the Apple II UCSD system. The Monitor's globals are hidden beneath the heap, and the Monitor code itself sits above your code space, so both are somewhat protected from inadvertent destruction.

I/O ERRORS

0	No error
1	Bad Block (Parity error)
2	Bad device number
3	Bad mode (Illegal operation)
4	Undefined hardware error
5	Lost device
6	Lost file
7	Bad file name
8	No room
9	No device
10	No file
11	Duplicate file
12	File not closed before open.
13	File not open
14	Bad format
15	Ring buffer overflow
16	Write-protect error
64	Device error

LOADER ERRORS

0	Unknown segment
1	No room in memory
2	Bad block
3	Can't read code file
4	Jump table overflow
5	SetSP at wrong place (after a physical link)
6	This loader does not handle intrinsic units
7	Too many units
8	Bad unit number
9	No INTRINSIC.LIB file
10	No unit location table
11	No segment location table
12	Cannot open intrinsic segment file
13	Cannot read file names block

"LisaPascalDevManFeb82_101.PICT" 57 KB 2000-12-12 dpi: 72h x 72v pix: 1321h x 1950v

FATAL ERRORS

0	Illegal index at trap handler
1	Stack Overflow
2	Programmed Halt
3	Range value error
4	Illegal string index
5	Can't read Root Volume

"LisaPascalDevManFeb82_102.PICT" 17 KB 2000-12-12 dpi: 72h x 72v pix: 1307h x 1965v

PASCAL COMPILER ERRORS

Lexical Errors:

```
  10  Too many digits
  11  Digit expected after '.' in real
  12  Integer overflow
  13  Digit expected in exponent
  14  End of line encountered in string constant
  15  Illegal character in input
  16  Premature end of file in source program
  17  Extra characters encountered after end of program
  18  End of file encountered in a comment
```

Syntactic Errors:

```
  20  Illegal symbol
  21  Error in simple type
  22  Error in declaration part
  23  Error in parameter list
  24  Error in constant
  25  Error in type
  26  Error in field list
  27  Error in factor
  28  Error in variable
  29  Identifier expected
  30  Integer expected
  31  '(' expected
  32  ')' expected
  33  '[' expected
  34  ']' expected
  35  ':' expected
  36  ';' expected
  37  '=' expected
  38  ',' expected
? 39  '*' expected
  40  ':=' expected
  41  'program' expected
  42  'of' expected
  43  'begin' expected
  44  'end' expected
  45  'then' expected
  46  'until' expected
  47  'do' expected
  48  'to' or 'downto' expected
? 49  'file' expected
? 50  'if' expected
  51  '.' expected
  52  'implementation' expected
  53  'interface' expected
  54  'intrinsic' expected
  55  'shared' expected
```


"LisaPascalDevManFeb82_103.PICT" 51 KB 2000-12-12 dpi: 72h x 72v pix: 1327h x 1961v

Semantic Errors:

```
  100  Identifier declared twice
  101  Identifier not of the appropriate class
  102  Identifier not declared
  103  Sign not allowed
  104  Number expected
  105  Lower bound exceeds upper bound
  106  Incompatible subrange types
  107  Type of constant must be integer
  108  Type must not be real
  109  Tagfield must be scalar or subrange
  110  Type incompatible with with tagfield type
  111  Index type must not be real
  112  Index type must be scalar or subrange
  113  Index type must not be 'integer' or 'longint'
  114  Unsatified forward reference
  115  Forward reference type identifier cannot appear in variable
       declaration
  116  Forward declaration - repetition of parameter list not allowed
  117  Forward declared function - repetition of result type not allowed
  118  Function result type must be scalar, subrange, or pointer
  119  File value parameter not allowed
  120  Missing result type in function declaration
  121  F-format for real only
  122  Error in type of standard function parameter
  123  Error in type of standard procedure parameter
  124  Number of parameters does not agree with declaration
  125  Illegal parameter substitution
? 126  Result type of parameteric function function does not agree with
       declaration
  127  Expression is not of set type
  128  Only tests on equality allowed
  129  Strict inclusion not allowed
  130  File comparison not allowed
  131  Illegal type of operand(s)
  132  Type of operand must be Boolean
  133  Set element type must be scalar or subrange
  134  Set element types not compatible
  135  Type of variable is not array or string
  136  Index type is not compatible with declaration
  137  Type of variable is not record
  138  Type of variable must be file or pointer
  139  Illegal type of loop control variable
  140  Illegal type of expression
  141  Assignment of files not allowed
  142  Label type incompatible with selecting expression
  143  Subrange bounds must be scalar
  144  Type conflict of operands
  145  Assignment to standard function is not allowed
? 146  Assignment to formal function is not allowed
  147  No such field in this record
? 148  Type error in read
```


"LisaPascalDevManFeb82_104.PICT" 87 KB 2000-12-12 dpi: 72h x 72v pix: 1313h x 1961v

```
  149  Actual parameter must be a variable
  150  Multidefined case label
  151  Missing corresponding variant declaration
? 152  Real or string tagfields not allowed
  153  Previous declaration was not forward
? 154  Substitution of standard proc/func is not allowed
  155  Multidefined label
  156  Multideclared label
  157  Undefined label
  158  Undeclared label
  159  Value parameter expected
  160  Multidefined record variant
? 161  File not allowed here
  162  Unknown compiler directive (not 'external' or 'forward')
  163  Variable cannot be packed field
  164  Set of real is not allowed
  165  Fields of packed records cannot be var parameters
  166  Case selector expression must be scalar or subrange
  167  String sizes must be equal
  168  String too long
  169  Value out of range
  170  Address of standard procedure cannot be taken
  171  Assignment to function result must be done inside that function
  172  Loop control variable must be local

  190  No such unit in this file

Conditional Compilation:

  260  New compile-time variable must be declared at global level
  261  Undefined compile-time variable
  262  Error in compile-time expression
  263  Conditional compilation options nested too deeply
  264  Unmatched ELSEC
  265  Unmatched ENDC
  266  Error in SETC
  267  Unterminated conditional compilation option


Compiler Specific Limitations:

  300  Too many nested record scopes
  301  Set limits out of range
  302  String limits out of range
  303  Too many nested procedures/functions
  304  Too many nested include/uses files
  305  Includes not allowed in interface section
  306  Pack and unpack are not implemented
  307  Too many units
  308  Set constant out of range

  350  Procedure too large
  351  File name in option too long
```


"LisaPascalDevManFeb82_105.PICT" 72 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1958v

I/O Errors:

400	Not enough room for code file
401	Error in rereading code file
402	Error in reopening text file
403	Unable to open uses file
404	Error in reading uses file
405	Error in opening include file
406	Eror in rereading previously read text block
407	Not enough room for i-code file
408	Error in writing code file
409	Error in reading i-code file
410	Unable to open listing file
420	I/O error on debug file

Code Generation Errors:

1000+	Code generator errors - should never occur
2000	End of I-code file not found
2001	Expression too complicated, code generator ran out of registers
2002	Code generator tried to free a register that was already free
2003-2005	Error in generating address
2006-2010	Error in expressions
2011	Too many globals
2012	Too many locals

Verification Errors:

4000	Bad verification block format
4001	Source code version conflict
4002	Compiler version conflict
4003	Linker version conflict
4100	Version in file less than minimum version supported by program
4101	Version in file greater than maximum version supported by program

"LisaPascalDevManFeb82_106.PICT" 50 KB 2000-12-12 dpi: 72h x 72v pix: 1323h x 1980v

ASSEMBLER ERRORS

0.
1. undefined label
2. operand out of range
3. must have procedure name
4. number of parameters expected
5. extra garbage on line
6. input line over 80 characters
7. not enough .IF's
8. must be declared in .ASECT before used
9. identifier previously declared
10. improper format
11. .EQU expected
12. must .EQU before use if not to a label
13. macro identifier expected
14. word addressed machine
15. backward .ORG currently not allowed
16. identifier expected
17. constant expected
18. invalid structure
19. extra special symbol
20. branch too far
21. variable not PC relative
22. illegal macro parameter index
23. not enough macro parameters
24. operand not absolute
25. illegal use of special symbols
26. ill-formed expression
27. not enough operands
28. cannot handle this relative expression
29. constant overflow
30. illegal decimal constant
31. illegal octal constant
32. illegal binary constant
33. invalid key word
34. macro stack overflow - 5 nested limit
35. include files may not be nested
36. unexpected end of input
37. this is a bad place for an .INCLUDE file
38. only labels & comments may occupy col 1
39. expected local label
40. local label stack overflow
41. string constant must be on one line
42. string constant exceeds 80 characters
43. illegal use of macro parameter
44. no local labels in .ASECT
45. expected key word
46. string expected
47. bad block, parity error (CRC)
48. bad unit number
49. bad mode, illegal operation
50. undefined hardware error
51. lost unit, unit is no longer on-line

"LisaPascalDevManFeb82_107.PICT" 66 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 1995v

```
 52.  lost file, file is no longer in directory
 53.  bad title, illegal file name
 54.  no room, insufficient space on disk
 55.  no unit, no such volume on-line
 56.  no file, no such file on volume
 57.  duplicate file
 58.  not closed, attempt to open an open file
 59.  not open, attempt to access a closed fil
 60.  bad format, error in reading real or int
 61.  nested macro definitions illegal
 62.  '=' or '<>' expected
 63.  may not EQU to undefined labels
 64.  must declare .ABSOLUTE before 1st .PROC
 65.
 66.
 67.
 68.
 69.
 70.  Not even a register
 71.  Not a Data Register.
 72.  Not an Address Register
 73.  Register Expected
 74.  Right Paren Expected
 75.  Right Paren or Comma Expected
 76.  Unrecognizable Operand
 77.  Odd location counter
 78.  Unimplemented Motorola directive
 79.  Comma Expected.
 80.  One operand must be a data register.
 81.  Dn,Dn or -(An),-(An) expected.
 82.  No longs allowed.
 83.  First operand must be immediate.
 84.  First operand must be Dn or #E
 85.  (An+),(An+) expected
 86.  Second operand must be an An
 87.  Second operand must be a Dn
 88.  #<data>,Dn expected.
 89.  first operand must be a Dn.
 90.  An,#<displacement> expected
 91.  An is not allowed with byte
 92.  only alterable addressing modes allowed
 93.  only data alterable addr modes allowed
 94.  An is not allowed
 95.  USP, SR, and CCR not allowed
 96.  Cannot move from CCR
 97.  Dx,d(Ay) or d(Ay),Dx expected.
 98.  Only memory alterable addr modes allowed
 99.  Only control addressing modes allowed
100.  Must branch backwards to label
```


FINIS

"LisaPascalDevManFeb82_108.PICT" 67 KB 2000-12-12 dpi: 72h x 72v pix: 1306h x 2008v